Current software frameworks in cognitive robotics integrating different computational paradigms

Hild, Manfred; Hülse, Martin

Publication date:
2008

Full-day Workshop on

Current software frameworks in cognitive robotics integrating different computational paradigms

in conjunction with
IEEE/RSJ 2008 International Conference on Intelligent RObots and Systems
September, 22-26, 2008, Nice, France
(IROS 2008)

Organizers: Martin Hülse and Manfred Hild

Table of Contents

A brief introduction to Current software frameworks in cognitive robotics integrating different computational paradigms

Martin Hülse and Manfred Hild

I. Motivation

Sophisticated robot systems have become an important part in cognition research. On the one hand, cognition research turned out to be a source of inspiration as well as a guidance to overcome current limitations in engineering of more complex and adaptive systems. On the other hand, cognition research projects have been utilizing robot systems as demonstrators and therefore they serve as an important proof of concept in this field. Furthermore, embodied cognition, in particular, is focused on the crucial role the body has for the development of cognitive behavior and therefore it becomes rather usual that experiments in this field of research involve robot systems of arbitrary complexity.

In order to create systems beyond the current state-of-the-art, engineers and scientists from different fields have to combine their approaches and know-how. One important area of this combination is software development and integration, where particular attention must be drawn to the different paradigms of scientists in cognition research and engineers [1].

An engineer creates systems, the component functions of which are most efficient when they meet a detailed set of specifications exactly. The consequence is high performance for a very specific task. But as soon as the application domain is extended or becomes more general a decline of performance must be expected.

Scientists in cognition research, and actually higher-level robotic applications are no exception, develop their programs, models and experiments in a language grounded in an ontology based on general principles. Hence, they expect reasonable and scalable performance for general domains and problem spaces.

For robotic related cognition research projects it is therefore essential to provide a robust and efficient software framework bridging lower-level services of heterogeneous robotic systems and high-level cognitive models.

II. Objectives

The objective of this workshop is to highlight and discuss existing examples of software frameworks combining autonomous robot systems and cognition research. We are targeting systems able to combine high-level cognitive models and heterogeneous robot system in a direct and robust manner. We are also particular interested in frameworks that allow the integration of different paradigms of computation intelligence, e.g. declarative languages and/or artificial neural networks.

The workshop is intended as a platform for exchange and discussion which might lead to the identification of general problems and possible solutions to be considered in future software design for complex robot systems. In the long run this might help in developing standards providing a direct knowledge transfer and exchange between cognitive system research, industry and application developers.

III. Topics

- existing software frameworks able to integrate different paradigms of computational intelligence for autonomous robot control
- general software architectures explicitly dealing with the integration of robot hardware and cognitive models
- frameworks for sustainable software design for heterogeneous robot platforms
- software frameworks providing implementations of cognitive models grounded in specific computational paradigms, but which are proven to be open for different robot platforms, i.e. open for different sensor and actuator systems

IV. Audience

Audience of this workshop is intended to be researchers as well as hardware and software engineers involved in ongoing or future middle and large robotic projects and who are interested in well proven and established software frameworks able to integrate heterogeneous robot hardware and which are open to arbitrary computational paradigms. We also welcome participants to share their experience of past robotic projects during our open discussion. In this sense, the workshop is open for people with a background from academia or industry.

References

[1] A. Farinelli, G. Grisetti and L. Iocchi, "Design and implementation of modular software for programming mobile robots" *Int. J. of Advanced Robotic Systems* vol. 3, 2006, pp. 37-42

M. Hülse, Department of Computer Science, Aberystwyth University, Aberystwyth, UK, msh@aber.ac.uk

M. Hild, Department of Computer Science Humboldt-Universität Berlin, Germany, hild@informatik.hu-berlin.de

Prototyping Cognitive Models with MARIE

Carle Côté, Patrick Frenette, Roger Champagne, François Michaud

***Abstract*— Since 2003, MARIE (Mobile and Autonomous Robotics Integration Environment) has been used to realize many robotic projects, ranging from a socially interactive autonomous mobile robot that must attend scientific conferences, to a mobile robot for telepresence in homes. MARIE's main objective is to integrate and combine heterogeneous software and computational paradigms in order to prototype various cognitive models applied to robotics. In this paper, we discuss conceptual and technical issues that must be addressed by component-based architectures like MARIE to support roboticists in their work.**

I. Introduction

In the last decades, with the venue of fields of study such as computational neuroscience, cybernetics and artificial intelligence, remarkable progresses have been made in the understanding of what is required to create artificial life evolving in real-world environments. Still, one of the remaining challenges is to create new cognitive models that would replicate high-level cognitive capabilities such as perception and information processing, reasoning, planning, learning and adaptation to new situations.

Lately, with accessibility to new technologies, robots are more frequently used as embodied systems to validate cognitive models. Unfortunately, implementation of those cognitive models usually requires wide expertise in many fields of study like probabilistic navigation, simultaneous localization and mapping, planning, speech recognition, audio and vision processing, etc. Moreover, cognitive models are derived from a large spectrum of computational paradigms that are not necessarily compatible when considering underlying software architecture requirements. This is why roboticists want to develop software frameworks that help deal with the integration of cognitive models requirements with software and hardware engineering methodologies and techniques.

MARIE (Mobile and Autonomous Robotics Integration Environment) [1], [2] is one attempt to create a flexible and versatile software integration environment adapted to prototyping cognitive models. It is based on a distributed component-based framework oriented towards integration and combination of heterogeneous software and computational paradigms. In this paper, we present two robotics projects developed with MARIE, illustrating some of the conceptual and technical issues that must be addressed to prototype different cognitive models.

Carle Côté, Patrick Frenette and François Michaud are with the Department of Electrical Engineering and Computer Engineering, Université de Sherbrooke, Sherbrooke, Québec, CANADA (e-mail: {Carle.Cote, Patrick.Frenette, Francois.Michaud}@USherbrooke.ca). F. Michaud holds the Canada Research Chair in Mobile Robotics and Intelligent Autonomous Systems.

Roger Champagne is with the Department of Software and IT Engineering, École de technologie supérieure, Université du Québec, Montréal, Québec, CANADA (e-mail: Roger.Champagne@ele.etsmtl.ca).

II. MARIE

MARIE is a distributed component-based middleware framework for robotic systems. To address the technical issues related to integration and combination of heterogeneous software and computational paradigms, multiple design solutions were developed to obtain a flexible framework that can be adapted to different scenarios.

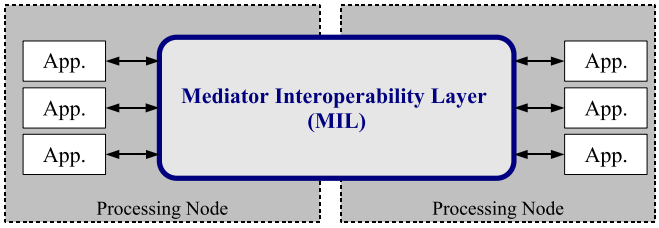


Fig. 1. Mediator Interoperability Layer (MIL)

A. Applications Mediation Approach

To implement distributed robotic systems using heterogeneous applications and computational paradigms, MARIE adapted the *Mediator* design pattern [3] to create a Mediator Interoperability Layer (MIL), illustrated in Fig 1. The *Mediator* design pattern primarily creates a centralized control unit (named Mediator) interacting with each component independently, and coordinating global interactions between applications to build the desired system. In MARIE, the MIL acts just like the *Mediator* design pattern, but is implemented as a virtual space where applications can interact together using a common language (similar to the relation between Internet and HTML for example). Note that the use of a virtual space implies that there is no single implementation class of the Mediator, as represented in the original pattern. The Mediator is distributed between all the applications that are linked together through the MIL, decentralizing the MIL's functionalities and responsibilities.

With the mediation approach, it is possible to create bridges between incompatible applications by having specialized code adapting each of them through the MIL. This way, each application can have its own communication protocols and mechanisms, as long as the MIL supports them and can bridge the application with others. For the robotics community, this approach offers a way to exploit the diversity of communication protocols and mechanisms, to benefit from

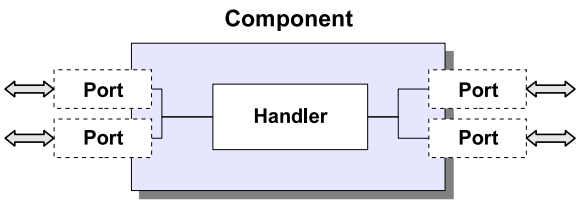


Fig. 2. Component Framework

their strengths and maximize their usage, and to overcome the lack of standards for robotic software system design.

B. Application Adapter & Communication Adapter

Existing applications do not necessarily implement the mechanisms, expose the interfaces or use a communication protocol that would make them compatible with the MIL. Changing an application's code to add the required functionalities must be avoided whenever possible. Instead, it is often preferable to use a wrapper technique to create a component which is compatible with the MIL, extending the application's functionalities without direct modifications. The main role of the wrapper component is to translate application service interfaces to make them compatible with the MIL's interface.

In MARIE, wrapper components used to encapsulate applications are called Application Adapters (AA), and wrapper components used to interconnect incompatible applications are called Communication Adapters (CA). To create an AA or CA, MARIE offers a development framework called the Component Framework, illustrated in Fig. 2. The *Handler* is responsible for the translation between application interfaces and the MIL interface. *Ports* are used to communicate with other components through the MIL by implementing and handling communication protocols (TCP/IP socket, UDP socket, Shared Memory, CORBA, IPC, COM, etc) through a simple abstraction interface supported by the Component Framework.

Although the use of MARIE's frameworks and software tools is highly encouraged to save time and efforts, MARIE is not limited to them. Developers can use the best solution to integrate software applications and interconnect components by having the possibility to extend or adapt existing components and available frameworks. MARIE's underlying philosophy is to complement existing applications, programming environments or software tools, and therefore it is to be used only when required and appropriate.

III. SPARTACUS

Spartacus [4] is a socially interactive mobile robot designed to enter the AAAI Mobile Robot Challenge, which consists of making a robot attend a conference just like humans. The robot has to navigate and localize itself in the world autonomously, extract visual information (such as reading messages, tracking people), localize, track and separate sound sources for enhanced speech recognition and dialogue interaction, provide graphical information through its touch screen interface, schedule tasks on its own, and give a presentation.

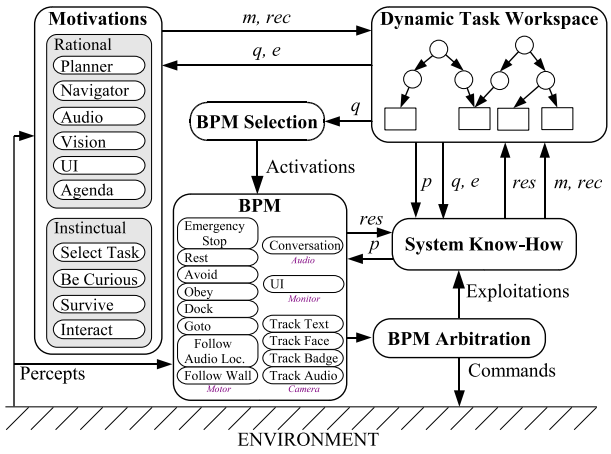


Fig. 3. MBA's implementation for Spartacus.

A. Cognitive Model

The cognitive model use to realize this project is called MBA (Motivated Behavioral Architecture) [4]. Figure 3 illustrates MBA's implementation for Spartacus. MBA is a behavior-based architecture that uses motivations to determine which available behavior-producing module(s) (BPM, or behaviors) should be activated at a specific time to control the robot. Motivation Modules (MM) are concurrent specialized algorithmic modules that recommend and monitor tasks (and/or sub-tasks) to execute in the Dynamic Task Workspace (DTW). Instinctual motivations provide basic operation of the robot and Rational motivations are more related to cognitive processes, such as navigation and planning. Motivations are kept independent from each other and do not necessarily share the same domain representation objects, except for the DTW tasks language. At a certain time interval, the BPM Selection module evaluates the tasks decomposition tree in the DTW and chooses which tasks/subtasks should be executed at that time. Once the task selection phase is completed, the System-Know-How module (SNOW) makes the correspondence between the selected tasks/subtasks and the BPMs available on the robot platform. Then, the BPM Arbitration module is executed on activated BPMs to ensure that coherent activations are applied (e.g., under a subsumption-based behavioral arbitration scheme, BPM Arbitration does not allow a move forward behavior activation concurrent to an avoid obstacle behavior activation). Then, using BPMs feedback information and perception from environment, DTW and Motivations are updated and the decision-making process is repeated.

B. Implementation using MARIE

While implementing MBA for Spartacus using MARIE, one of our first objectives was to reuse software packages already available, such as:

- **Player/Stage**[1] is a project to create free software for robotics and sensor systems.
- **FlowDesigner/RobotFlow**. FlowDesigner[2] is a free data flow oriented development environment, with RobotFlow[3] being the mobile robotics toolkit for FlowDesigner.
- **CARMEN Navigation Software**[4] is a software package for laser-based autonomous navigation using previously generated maps.
- **Pmap Map Builder Software**[5] provides libraries and utilities for laser-based mapping in 2D environments to produce high-quality occupancy grid maps.
- **Vision Processing Library** is used to extract symbols and text from a single color image in real world conditions [5].
- **OpenCV Computer Vision Library**[6] is an open source computer vision library.
- **Festival**[7] offers a general framework for building speech synthesis systems.
- **Sound Processing Library** is for the localization, tracking and separation of sound sources using a microphone array system [6].
- **Nuance**[8] is a commercial speech recognition software.
- **QT3**[9] is a cross-platform application development framework, used for the development of our GUIs.

Software integration of all these elements was not a simple plug-and-play process. Most of them were developed independently, adopting different design and implementation requirements. Moreover, they had to be used in software modules following MBA principles. Spartacus' software architecture is illustrated in Figure 4.

Spartacus's software architecture provides two setups. For the Real Robot setup, SpartacusAA combines wheels odometry with gyroscopic data, and pushes the result at a fixed rate (200 Hz) to its interconnected components. Laser data is collected by PlayerAA, interfacing with the Player library specialized for sensor and actuator abstraction [7], supporting the SICK LMS200 laser range finder. PlayerAA pushes data at a fixed rate (200 Hz) to connected components. For the Simulation setup, odometry and laser data are both collected with PlayerAA, generated either by Stage (2D) or Gazebo (3D) simulators [7]. CARMEN Localizer AA and CARMEN Path Planner AA provide path planning and localization capabilities. CARMEN is composed of small processes communicating through a central server. CARMEN's integration was realized by creating an AA that starts several of these processes depending on the required functionality and on data conversion from CARMEN's to MARIE's format.

[1]http://playerstage.sourceforge.net/
[2]http://flowdesigner.sourceforge.net/
[3]http://robotflow.sourceforge.net/
[4]http://carmen.sourceforge.net/
[5]http://robotics.usc.edu/~ahoward/pmap
[6]http://www.intel.com/technology/computing/opencv/index.htm
[7]http://www.cstr.ed.ac.uk/projects/festival/
[8]http://www.nuance.com
[9]http://www.trolltech.com/products/qt

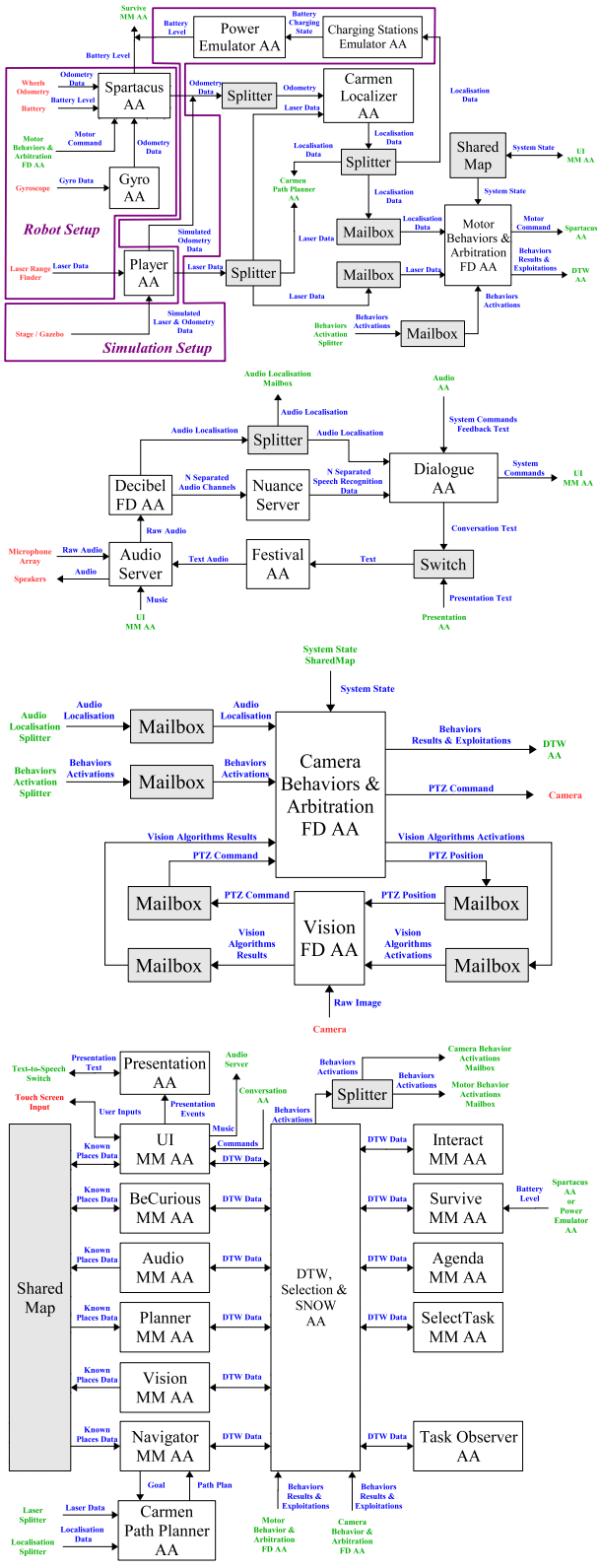


Fig. 4. Spartacus' software architecture implemented using MARIE.

FlowDesigner and RobotFlow are used to implement Behavior & Arbitration FD AA, handling BPMs and their arbitration. FlowDesigner uses a synchronous pull mechanism to get data coming from different elements such as localization, path plan, laser, audio localization, dialogue command and system states, requiring the use of Mailbox CA components. By buffering input data, mailboxes allows AAs running at different rates to be interconnected. Behavior & Arbitration FD AA generates motor commands at a fixed rate (5 Hz).

The Audio Server is interfacing a RME Hammerfal DSP Multiface sound card, and NUANCE Server is interfacing NUANCE. DialogueAA is a stand-alone AA that manages simultaneous conversations with people. This is made possible with the use of AUDIBLE FD AA, interfacing our sound source localization, tracking and separation algorithms implemented with RF/FD and using Spartacus' microphone array. It generates a number of separated audio channels that are sent to NUANCE Server and Behavior & Arbitration FD AA. Integrating NUANCE in an AA was challenging since it is a proprietary application with a fixed programming interface, and because its execution flow is tightly controlled by NUANCE's core application, which is not accessible from the available interface. To solve this problem, we created an independent application that uses a communication protocol already supported by MARIE. Recognized speech data is sent to Dialogue AA, responsible of the human-robot vocal interface. Speech generated by the robot is handled by Festival [8]. The Dialogue AA conversation context mode is selected by the Audio MM AA, monitoring the tasks present in the DTW and requiring speech interaction.

The global execution of the system is asynchronous, having most of the applications and AAs pushing their results at variable rates (determined by the computation length of their algorithms when triggered by new input data). Synchronous execution is realized by having fixed rate sensor readings and actuator command writings.

Overall, Spartacus' implementation with MARIE required 42 components (approx. 50 000 lines of code) composed of 26 AAs, 14 CAs and two external applications (the Audio Server and NUANCE). To get enough processing, multiple processors were required : vision and audio components were distributed on two on-board laptop computers, all components related to the decision-making architecture and GUIs were executed on Spartacus' on-board Mini-ITX computer, and finally all teleoperation controls required to manipulate the robot safely were executed on a remote laptop using wireless communication with the platform. Distributing applications and adapters across multiple processing nodes was pretty straightforward with MARIE by choosing socket-based Push, Pull and Events dataflow communication mechanisms [9] for each adapter's port. All communication protocols use XML encoding for data representation, except for the Audio Server and NUANCE which use their own communication protocols.

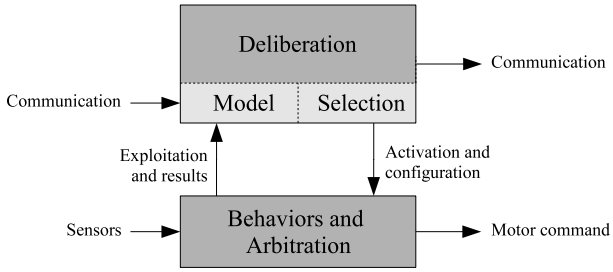


Fig. 5. Dynamic Platooning cognitive model

IV. Dynamic Platooning

The objective of the Dynamic Platooning project is to coordinate maneuvers of autonomous vehicles inside a platoon (i.e., insertion, exiting, collision avoidance, emergency stop, emergency exit, etc.). A platoon is formed by a set of vehicles following each other at a very close distance. Inter-robot communication is used to signal intentions of executing any maneuver to other vehicles or to signal emergency situations like accidents, maneuver execution failures and obstacles to avoid.

A. Cognitive Model

The cognitive model used to realize the project, illustrated in Figure 5, is a distributed hybrid architecture. The decision-making architecture is decomposed as follow. The *Deliberative layer* manages communications with other vehicles and keeps a local model of the platoon. It also evaluates which maneuver should be executed using a finite state machine (FSM). The *Intermediate layer* applies a selection function to activate and configure the behaviors according to the FSM states. The *Behaviors and Arbitration layer* uses a subsumption mechanism to select which activated behavior will produce the resulting motor command. Behaviors always receive Sensors information to react to meaningful changes in the environment. Behaviors Exploitation and results are used as feedback information injected to the Deliberative layer to update the local platoon model and influence the decision-making process.

B. Implementation using MARIE

Figure 6 shows a simplified illustration of MARIE's components used to implement the cognitive model used by each robot. We used a group of four Pioneer II robots in our experiments.

As for the Spartacus project, reusing available software packages was important. AA for Player and FlowDesigner were already available at that time. Therefore, we reused them without any modification. We configured Player AA to provide the odometry and sonar range data to the behaviors and to output the motor commands given by the decision-making system to the vehicle actuators. FlowDesigner AA was configured to interface a sensor that detects and positions each vehicle in the environment relative to each other [10].

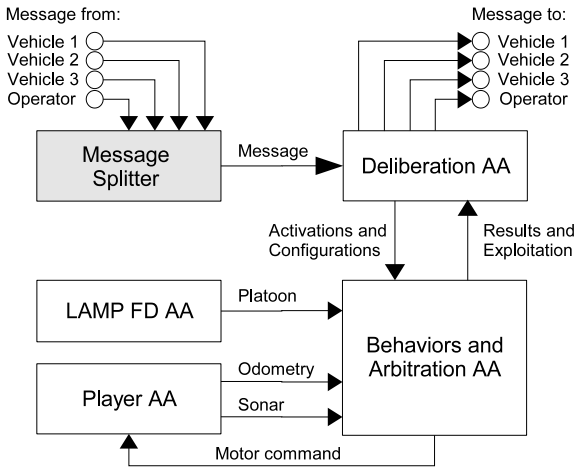


Fig. 6. Dynamic Platooning distributed software architecture implemented using MARIE.

To implement the *Behaviors and Arbitration layer*, a generic behavior component distributed in MARIE's components library was used. This component, called Behaviors AA, makes it possible to customize the sensory inputs, to map the inputs to the behaviors and to assign priority to the behaviors. The addition of specific behaviors required for the project was simple since each behavior was developed using a simple software interface allowing them to be dynamically loaded by the AA.

The *Deliberative layer* was entirely built within a custom AA called the Deliberation AA. It contains the platoon model, the communication management system and the selection function, all coded in C++. The FSM was developed using an external library called the State Map Compiler[10] that generates C++ code, also integrated within the Deliberation AA.

An important application, called Operator, was also required to control experimentations by initiating scenarios like start a maneuver, simulate an accident or simulate a software failure. Operator was developed in Java, and it was designed to run on a control laptop communicating with each platoon member using a wireless communication system. All communications coming from other platoon members or the Operator application were routed to the *Deliberative layer* using Message Splitters.

Overall, the implementation with MARIE required eight components per vehicle (5 AAs, 3 CAs) and five components on the control laptop (1 AA, 3 CAs and 1 Java application that monitor each trial) for a total of 37 components. The AM was used to distribute the process on each vehicle and the control laptop. The 36 configuration files were on the control laptop and allowed easy modification of the files. As for Spartacus, socket-based Push, Pull and Events dataflow communication mechanisms [9] were used in each adapter's port. Communication protocols all used XML encoding.

[10]http://smc.sourceforge.net/

V. Conclusion

Our experience with MARIE shows that developing a software integration environment adapted to prototyping various cognitive models requires a flexible and versatile design. It should allow heterogeneous software reuse and support different solutions to solve integration issues. It should offer extensible and configurable software modules to adapt to different implementation scenarios. It should also give access to a set of useful tools that are required by most implementations such as component deployment tools on multiple processing nodes, a configurable logging system for debugging and data analysis, and data visualization tools to help understand the dynamics of implemented systems.

Future work includes more in-depth studies of the impact of overhead introduced by MARIE when developing and deploying systems, and how to achieve stability and robustness when having to support a large spectrum of computational paradigms and heterogeneous software applications.

MARIE is available as an open source project at http://marie.sourceforge.net.

VI. Acknowledgements

The authors gratefully acknowledge the contribution of the Canada Research Chair (CRC), the Natural Sciences and Engineering Research Council of Canada (NSERC), the Canadian Foundation for Innovation (CFI) and the Network of Centres of Excellence on the Automobile of the 21st Century (AUTO21) in the support of this work.

References

[1] C. Cote, Y. Brosseau, D. Letourneau, C. Raievsky and F. Michaud, "Robotic Software Integration Using MARIE", *Int. Journal of Advanced Robotic Systems*, vol. 3, no. 1, pp. 55-60, Mar. 2006.

[2] C. Cote, D. Letourneau, F. Michaud and Y. Brosseau, *Using MARIE for Mobile Robot Component Development and Integration*, ser. Springer Tracts in Advanced Robotics : Principles and Practice of Software Development in Robotics. Springer-Verlag Heidelberg, 2007.

[3] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design patterns : Elements of reusable object-oriented software*. Reading, MA: Addison-Wesley, 1994.

[4] F. Michaud, Y. Brosseau, C. Côté, D. Létourneau, P. Moisan, A. Ponchon, C. Raievsky, J.-M. Valin, E. Beaudry and F. Kabanza, "Modularity and integration in the design of a socially interactive robot", in *Proceedings IEEE International Workshop on Robot and Human Interactive Communication*, 2005, pp. 172-177.

[5] D. Letourneau, F. Michaud and J.-M. Valin, "Autonomous robot that can read", *EURASIP Journal on Applied Signal Processing, Special Issue on Advances in Intelligent Vision Systems: Methods and Applications*, vol. 17, pp. 1-14, 2004.

[6] J.-M. Valin, F. Michaud and J. Rouat, "Robust 3D localization and tracking of sound sources using beamforming and particle filtering", in *Proceedings International Conference on Audio, Speech and Signal Processing*, 2006, pp. 221-224.

[7] R. T Vaughan, B. P. Gerkey and A. Howard, "On device abstractions for portable, reusable robot code", in *Proceedings IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2003, pp. 2421-2427.

[8] P. Taylor, "The Festival speech architecture", URL: http://www.cstr.ed.ac.uk/projects/festival/, 1999.

[9] Y. Zhao, "A model of computation with push and pull processing", Master's thesis, University of California at Berkeley, Department of Electrical Engineering and Computer Science, 2003.

[10] F. Rivard, J. Bisson, F. Michaud and D. Ltourneau, "Ultrasonic relative positioning for multi-robot systems", *IEEE International Conference on Robotics and Automation*, 2008.

Bridging the Sense-Reasoning Gap: DyKnow - A Middleware Component for Knowledge Processing

Fredrik Heintz and Jonas Kvarnström and Patrick Doherty
Department of Computer and Information Science, Linköpings universitet,
SE-581 83 Linköping, Sweden
{frehe, jonkv, patdo}@ida.liu.se

***Abstract*— Developing autonomous agents displaying rational and goal-directed behavior in a dynamic physical environment requires the integration of both sensing and reasoning components. Due to the different characteristics of these components there is a gap between sensing and reasoning. We believe that this gap can not be bridged in a single step with a single technique. Instead, it requires a more general approach to integrating components on many different levels of abstraction and organizing them in a structured and principled manner.**

In this paper we propose knowledge processing middleware as a systematic approach for organizing such processing. Desirable properties of such middleware are presented and motivated. We then go on to argue that a declarative stream-based system is appropriate to provide the desired functionality. Finally, DyKnow, a concrete example of stream-based knowledge processing middleware that can be used to bridge the sense-reasoning gap, is presented. Different types of knowledge processes and components of the middleware are described and motivated in the context of a UAV traffic monitoring application.

I. Introduction

When developing autonomous agents displaying rational and goal-directed behavior in a dynamic physical environment, we can lean back on decades of research in artificial intelligence. A great number of deliberative and reactive functionalities have already been developed, including chronicle recognition, motion planning, task planning and execution monitoring. To integrate these approaches into a coherent system it is necessary to reconcile the different formalisms used to represent information and knowledge about the world. To construct these world models and maintain a correlation between them and the environment it is necessary to extract information and knowledge from data collected by sensors. However, most research done in a symbolic context tends to assume crisp knowledge about the current state of the world while the information extracted from the environment often is noisy and incomplete quantitative data on a much lower level of abstraction. This causes a wide gap between sensing and reasoning.

Bridging this gap in a single step, using a single technique, is only possible for the simplest of autonomous systems. As complexity increases, one typically requires a combination of a wide variety of methods, including more or less standard functionalities such as various forms of image processing and information fusion as well as application-specific and possibly even scenario-specific approaches. Such integration is currently done ad hoc, partly by allowing the sensory and deliberative layers of a system to gradually extend towards each other and partly by introducing intermediate processing levels.

In this paper, we propose using the term *knowledge processing middleware* for a principled and systematic framework for bridging the gap between sensing and deliberation in a physical agent. We claim that knowledge processing middleware should provide both a conceptual framework and an implementation infrastructure for integrating a wide variety of components and managing the information that needs to flow between them. It should allow a system to incrementally process low-level sensor data and generate a coherent view of the environment at increasing levels of abstraction, eventually providing information and knowledge at a level which is natural to use in symbolic deliberative functionalities. Such a framework would also support the integration of different deliberation techniques.

The structure of the paper is as follows. In the next section, an example scenario is presented as further motivation for the need for a systematic knowledge processing middleware framework. Desirable properties of such frameworks are investigated and a specific stream-based architecture is proposed which is suitable for a wide range of systems. As a concrete example, our framework DyKnow is briefly described. The paper is concluded with some related work and a summary.

II. A Traffic Monitoring Scenario

Traffic monitoring is an important application domain for research in autonomous unmanned aerial vehicles (UAVs) which provides a plethora of cases demonstrating the need for an intermediary layer between sensing and deliberation. It includes surveillance tasks such as detecting accidents and traffic violations, finding accessible routes for emergency vehicles, and collecting statistics about traffic patterns.

In the case of detecting traffic violations, one possible approach relies on using a formal declarative description of each type of violation. This can be done using a chronicle [7],

This work is partially supported by grants from the Swedish Aeronautics Research Council (NFFP4-S4203), the Swedish Foundation for Strategic Research (SSF) Strategic Research Center MOVIII and the Center for Industrial Information Technology CENIIT (06.09).

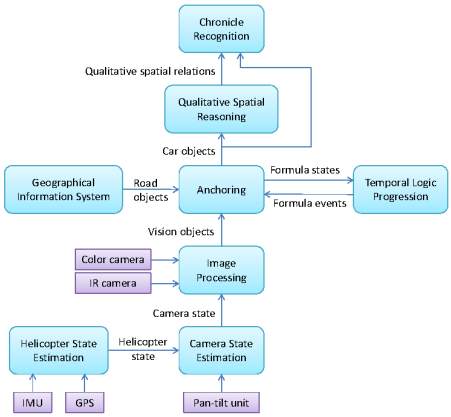


Fig. 1. An overview of how the incremental processing for the traffic surveillance task could be organized.

which defines a class of complex events using a simple temporal network where nodes correspond to occurrences of high level qualitative events and edges correspond to metric temporal constraints between event occurrences. For example, to detect a reckless overtake, qualitative spatial events such as $\mathit{beside}(\mathit{car}_1, \mathit{car}_2)$, $\mathit{close}(\mathit{car}_1, \mathit{car}_2)$ and $\mathit{on}(\mathit{car}_1, \mathit{road}_7)$ might be used. Creating such high-level representations from low-level sensory data, such as video streams from color and infrared cameras, involves a great deal of work at different levels of abstraction which would benefit from being separated into distinct and systematically organized tasks.

Figure 1 provides an overview of how the incremental processing required for the traffic surveillance task could be organized.

At the lowest level, a helicopter state estimation system uses data from inertial measurement unit and GPS sensors to determine the current position and attitude of the helicopter. The resulting information is fed into a camera state estimation system, together with the current angles of the pan-tilt unit on which the cameras are mounted, to generate information about the current camera state. The image processing system uses the camera state to determine where the camera is currently pointing. Video streams from the color and thermal cameras can then be analyzed in order to extract vision objects representing hypotheses regarding moving and stationary physical entities, including their approximate positions and velocities.

Each vision object must be associated with a symbol for use in higher level services, a process known as *anchoring* [5, 11]. In the traffic surveillance domain, identifying which vision objects correspond to vehicles is also essential, which requires knowledge about normative sizes and behaviors of vehicles. One interesting approach to describing such behaviors relies on the use of formulas in a metric temporal modal logic, which are incrementally progressed through states that include current vehicle positions, velocities, and other relevant information. An entity satisfying all requirements can be hypothesized to be a vehicle, a hypothesis which is subject to being withdrawn if the entity ceases to satisfy the normative behavior and thereby causes the formula progression system to signal a violation.

As an example, vehicles usually travel on roads. Given that image processing provided absolute world coordinates for each vision object, the anchoring process can query a geographic information system to determine the nearest road segment and derive higher level predicates such as $\mathit{on\text{-}road}(\mathit{car})$ and $\mathit{in\text{-}crossing}(\mathit{car})$. These would be included in the states sent to the progressor as well as in the vehicle objects sent to the next stage of processing, which involves deriving qualitative spatial relations between vehicles such as $\mathit{beside}(\mathit{car}_1, \mathit{car}_2)$ and $\mathit{close}(\mathit{car}_1, \mathit{car}_2)$. These predicates, and the concrete events corresponding to changes in the predicates, finally provide sufficient information for the chronicle recognition system to determine when higher-level events such as reckless overtakes occur.

In this example, we can identify a considerable number of distinct processes involved in bridging the gap between sensing and deliberation and generating the necessary symbolic representations from sensor data. However, in order to fully appreciate the complexity of the system, we have to widen our perspective somewhat. Looking towards the smaller end of the scale, we can see that what is represented as a single process in Figure 1 is sometimes merely an abstraction of what is in fact a set of distinct processes. Anchoring is a prime example, encapsulating tasks such as the derivation of higher level predicates which could also be viewed as a separate process. At the other end of the scale, a complete UAV system also involves numerous other sensors and information sources as well as services with distinct knowledge requirements, including task planning, path planning, execution monitoring, and reactive procedures.

Consequently, what is seen in Figure 1 is merely an abstraction of the full complexity of a small part of the system. It is clear that a systematic means for integrating all forms of knowledge processing, and handling the necessary communication between parts of the system, would be of great benefit. Knowledge processing middleware should fill this role, by providing a standard framework and infrastructure for integrating image processing, sensor fusion, and other data, information and knowledge processing functionalities into a coherent system.

III. Knowledge Processing Middleware

Any principled and systematic framework for bridging the gap between sensing and deliberation in a physical agent qualifies as knowledge processing middleware, by the definition in the introduction. We now consider the necessary and desirable properties of any such framework.

The first requirement is that the framework should *permit the integration of information from distributed sources*, allowing this information to be processed at many different levels of abstraction and finally transformed into a suitable form to be used by a deliberative functionality. In the traffic monitoring scenario, the primary input will consist of low level sensor data such as images, a signal from

a barometric pressure sensor, a GPS (Global Positioning System) signal, laser range scans, and so on. But there might also be high level information available such as geographical information and declarative specifications of traffic patterns and normative behaviors of vehicles. The middleware must be sufficiently flexible to allow the integration of all these different sources into a coherent processing system. Since the appropriate structure will vary between applications, a general framework should be agnostic as to the types of data and information being handled and should not be limited to specific connection topologies.

To continue with the traffic monitoring scenario, there is a natural abstraction hierarchy starting with quantitative signals from sensors, through image processing and anchoring, to representations of objects with both qualitative and quantitative attributes, to high level events and situations where objects have complex spatial and temporal relations. Therefore a second requirement is the *support of quantitative and qualitative processing* as well as a mix of them.

A third requirement is that *both bottom-up data processing and top-down model-based processing should be supported*. Different abstraction levels are not independent. Each level is dependent on the levels below it to get input for bottom-up data processing. At the same time, the output from higher levels could be used to guide processing in a top-down fashion. For example, if a vehicle is detected on a particular road segment, then a vehicle model could be used to predict possible future locations, which could be used to direct or constrain the processing on lower levels. Thus, a knowledge processing framework should not impose strict bottom-up nor strict top-down processing.

A fourth requirement is support for *management of uncertainty* on different levels of abstraction. There are many types of uncertainty, not only at the quantitative sensor data level but also in the symbolic identity of objects and in temporal and spatial aspects of events and situations. Therefore it is not realistic to use a single approach to handling uncertainty throughout a middleware framework. Rather, it should allow many different approaches to be combined and integrated into a single processing system in a manner appropriate to the specific application at hand.

Physical agents acting in the world have limited resources, both in terms of processing power and in terms of sensors, and there may be times when these resources are insufficient for satisfying the requests of all currently executing tasks. In these cases a trade-off is necessary. For example, reducing update frequencies would cause less information to be generated, while increasing the maximum permitted processing delay would provide more time to complete processing. Similarly, an agent might decide to focus its attention on the most important aspects of its current situation, ignoring events or objects in the periphery, or to focus on providing information for the highest priority tasks or goals. An alternative could be to replace a resource-hungry calculation with a more efficient but less accurate one. Each trade-off will have effects on the quality of the information produced and the resources used. Another reason for changing the processing is that it is often context dependent and as the context changes the processing needs to change as well. For example, the processing required to monitor the behavior of vehicles following roads and vehicles which may drive off-road is very different. In the first case simplifying assumptions can be made as to how vehicles move, while these would be invalid if a vehicle goes off-road. To handle both cases a system would have to be able to switch between the different processing configurations. A fifth requirement on knowledge processing middleware is therefore support for *flexible configuration and reconfiguration* of the processing that is being performed.

An agent should not depend on outside help for reconfiguration. Instead, it should be able to reason about which trade-offs can be made at any point in time, which requires introspective capabilities. Specifically, the agent must be able to determine what information is currently being generated as well as the potential effects of any changes it may make in the processing structure. Therefore a sixth requirement is for the framework to provide a *declarative specification of the information being generated and the information processing functionalities that are available*, with sufficient content to make rational trade-off decisions.

To summarize, knowledge processing middleware should support declarative specifications for flexible configuration and dynamic reconfiguration of context dependent processing at many different levels of abstraction.

IV. Stream-Based Knowledge Processing Middleware

The previous section focused on requirements that are necessary or desirable in any form of knowledge processing middleware, intentionally leaving open the question of how these requirements should be satisfied. We now go on to propose one specific type of framework, *stream-based* knowledge processing middleware, which we believe will be useful in many applications. A concrete implementation, DyKnow, will be discussed later in this paper.

Due to the need for incremental refinement of information at different levels of abstraction, we model computations and processes within the stream-based knowledge processing framework as active and sustained *knowledge processes*. The complexity of such processes may vary greatly, ranging from simple adaptation of raw sensor data to image processing algorithms and potentially reactive and deliberative processes.

In our experience, it is not uncommon for knowledge processes at a lower level to require information at a higher frequency than those at a higher level. For example, a sensor interface process may query a sensor at a high rate in order to average out noise, providing refined results at a lower effective sample rate. This requires knowledge processes to be decoupled and asynchronous to a certain degree. In stream-based knowledge processing middleware, this is achieved by allowing a knowledge process to declare a set of *stream generators*, each of which can be *subscribed* to by an arbitrary number of processes. A subscription can be viewed as a continuous query, which creates a distinct asynchronous

stream onto which new data is pushed as it is generated. The contents of a stream may be seen by the receiver as data, information or knowledge.

Decoupling processes through asynchronous streams minimizes the risk of losing samples or missing events, something which can be a cause of problems in query-based systems where it is the responsibility of the receiver to poll at sufficiently high frequencies. Streams can provide the necessary input for processes that require a constant and timely flow of information. For example, a chronicle recognition system needs to be apprised of all pertinent events as they occur, and an execution monitor must receive constant updates for the current system state at a given minimum rate. A push-based stream system also lends itself easily to "on-availability" processing, i.e. processing data as soon as it is available, and the minimization of processing delays, compared to a query-based system where polling introduces unnecessary delays in processing and the risk of missing potentially essential updates as well as wastes resources. Finally, decoupling also facilitates the distribution of processes within a platform or between different platforms, another important property of many complex autonomous systems.

Finding the correct stream generator requires each stream generator to have an identity which can be referred to, a *label*. Though a label could be opaque, it often makes sense to use structured labels. For example, given that there is a separate position estimator for each vehicle, it makes sense to provide an identifier i for each vehicle and to denote the (single) stream generator of each position estimator by *position*$[i]$. Knowing the vehicle identifier is sufficient for generating the correct stream generator label.

Even if many processes connect to the same stream generator, they may have different requirements for their input. As an example, one could state whether new information should be sent "when available", which is reasonable for more event-like information or discrete transitions, or with a given frequency, which is more reasonable with continuously varying data. In the latter case, a process being asked for a subscription at a high frequency may need to alter its own subscriptions to be able to generate stream content at the desired rate. Requirements may also include the desired approximation strategy when the source knowledge process lacks input, such as interpolation or extrapolation strategies or assuming the previous value persists. Thus, every subscription request should include a *policy* describing such requirements. The stream is then assumed to satisfy this policy until it is removed or altered. For introspection purposes, policies should be declaratively specified.

While it should be noted that not all processing is based on continuous updates, neither is a stream-based framework limited to being used in this manner. For example, a path planner or task planner may require an initial state from which planning should begin, and usually cannot take updates into account. Even in this situation, decoupling and asynchronicity are important, as is the ability for lower level processing to build on a continuous stream of input before it can generate the desired snapshot. A snapshot query, then, is simply a special case of the ordinary continuous query.

A. Knowledge Processes

For the purpose of modeling, we find it useful to identify four distinct types of knowledge processes: Primitive processes, refinement processes, configuration processes and mediation processes.

Primitive processes serve as an interface to the outside world, connecting to sensors, databases or other information sources that in themselves have no explicit support for stream-based knowledge processing. Such processes have no stream inputs but provide a non-empty set of stream generators. In general, they tend to be quite simple, mainly adapting data in a multitude of external representations to the stream-based framework. For example, one process may use a hardware interface to read a barometric pressure sensor and provide a stream generator for this information. However, greater complexity is also possible, with primitive processes performing tasks such as image processing.

The remaining process types will be introduced by means of an illustrating example from the traffic monitoring scenario, where car objects must be generated and anchored to sensor data which is mainly collected using cameras. Note that this example is not purely theoretical but has been fully implemented and successfully used in test flights in an experimental UAV platform [13].

In the implemented approach, the image processing system produces *vision objects* representing entities found in an image, called *blobs*, having visual and thermal properties similar to those of a car. A vision object state contains an estimation of the size of the blob and its position in absolute world coordinates. When a new vision object has been found, it is tracked for as long as possible by the image processing system and each time it is found in an image a new vision object state is pushed on a stream.

Anchoring begins with this stream of vision object states, aiming at the generation of a stream of *car object* states providing a more qualitative representation, including relations between car objects and road segments. An initial filtering process, omitted here for brevity, determines whether to hypothesize that a certain vision object in fact corresponds to a car. If so, a car object is created and a *link* is established between the two objects. To monitor that the car object actually behaves like a car, a maintain constraint describing expected behavior is defined. The constraint is monitored, and if violated, the car hypothesis is withdrawn and the link is removed. A temporal modal logic is used for encoding normative behaviors, and a progression algorithm is used for monitoring that the formula is not violated.

Figure 2 shows an initial process setup, existing when no vision objects have been linked to car objects. As will be seen, processes can dynamically generate new processes when necessary. Figure 3 illustrates the process configuration when VisionObject#51 has been linked to CarObject#72 and two new refinement processes have been created.

The first process type to be considered is the *refinement process*, which takes a set of streams as input and provides

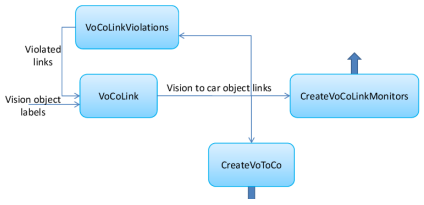


Fig. 2. The initial set of processes before any vision object has been created.

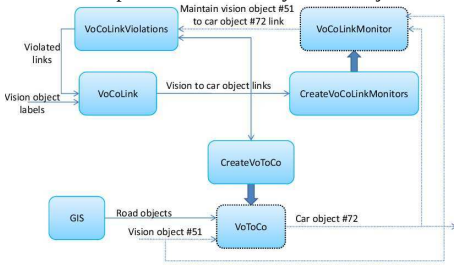


Fig. 3. The set of processes after VisionObject#51 has been linked to CarObject#72.

one or more stream generators producing refined, abstracted or otherwise processed values. Several examples can be found in the traffic monitoring application, such as:

- VoCoLink – Manages the set of links between vision objects and car objects, each link being represented as a pair of labels. When a previously unseen vision object label is received, create a new car object label and a link between them. When a link is received from the VoCoLinkViolations process, the maintain constraint of the link has been violated and the link is removed. The output is a stream of sets of links. A suitable policy may request notification only when the set of links changes.
- VoToCo – Refines a single vision object to a car object by adding qualitative information such as which road segment the object is on and whether the road segment is a crossing or a road. Because quantitative data is still present in a car object, a suitable policy may request new information to be sent with a fixed sample frequency. Using a separate process for each car object yields a fine-grained processing network where different cars may be processed at different frequencies depending on the current focus of attention.
- VoCoLinkMonitor – An instantiation of the formula progressor. Monitors the maintain constraint of a vision object to car object link, using the stream of car object states generated by the associated VoToCo. The output is false iff the maintain constraint has been violated.

The second type of process, the *configuration process*, takes a set of streams as input but produces no new streams. Instead, it enables dynamic reconfiguration by adding or removing streams and processes. The configuration processes used in our example are:

- CreateVoCoLinkMonitors – Takes a stream of sets of links and ensures VoCoLinkMonitor refinement processes are created and removed as necessary.
- CreateVoToCos – Takes a stream of vision to car object links and ensures VoToCo refinement processes are created and removed as necessary.

Finally, a *mediation process* generates streams by selecting or collecting information from other streams. Here, one or more of the inputs can be a stream of labels identifying other streams to which the mediation process may subscribe. This allows a different type of dynamic reconfiguration in the case where not all potential inputs to a process are known in advance or where one does not want to simultaneously subscribe to all potential inputs due to processing cost. One mediation process is used in our example:

- VoCoLinkViolations – Takes a stream of sets of links identifying all current connections between vision objects and car objects. Dynamically subscribes to and unsubscribes from monitor information from the associated VoCoLinkMonitors as necessary. If a monitor signals a violation (sending the value “false”), the corresponding link becomes part of the output, a stream of sets of violated links.

In Figure 2 the VoCoLinkViolations mediation process subscribes to no streams, since there are no VoCoLinkMonitor streams. In Figure 3 it subscribes to the stream of monitor results of the maintain constraint of the new VisionObject#51 to CarObject#72 link.

This example shows how stream-based knowledge processing middleware can be applied in a very fine-grained manner, even at the level of individual objects being tracked in an image processing context. At a higher level, the entire anchoring process can be viewed as a composite knowledge process with a small number of inputs and outputs, as originally visualized in Figure 1. Thus, one can switch between different abstraction levels while remaining within the same unifying framework.

B. Timing

Any realistic knowledge processing architecture must take into account the fact that both processing and communication takes time, and that delays may vary, especially in a distributed setting. As an example, suppose one knowledge process is responsible for determining whether two cars are too close to each other. This test could be performed by subscribing to two car position streams and measuring the distance between the cars every time a new position sample arrives. Should one input stream be delayed by one sample period, distance calculations would be off by the distance traveled during that period, possibly triggering a false alarm. Thus, the fact that two pieces of information arrive simultaneously must not be taken to mean that they refer to the same time.

For this reason, stream-based knowledge processing middleware should support tagging each piece of information in a stream with its *valid time*, the time at which the information was valid in the physical environment. For example, an image taken at time t has the valid time t. If an image processing system extracts vision objects from this image, each created vision object should have the same valid time even though

some time will have passed during processing. One can then ensure that only samples with the same valid time are compared. Valid time is also used in temporal databases [15].

Note that nothing prevents the creation of multiple samples with the same valid time. For example, a knowledge process could very quickly provide a first rough estimate of some property, after which it would run a more complex algorithm and eventually provide a better estimate with identical valid time.

The *available time*, the time when a piece of information became available through a stream, is also relevant. If each value is tagged with its available time, a knowledge process can easily determine the total aggregated processing and communication delay associated with the value, which is useful in dynamic reconfiguration. Note that the available time is not the same as the time when the value was retrieved from the stream, as retrieval may be delayed by other processing.

The available time is also essential when determining whether a system behaves according to specification, which depends on the information actually available at any time as opposed to information that has not yet arrived.

V. DyKnow

A concrete example of a stream-based knowledge processing middleware framework called DyKnow has been developed as part of our effort to build UAVs capable of carrying out complex missions [6, 10, 12]. Most of the functionality provided by DyKnow has already been presented in the previous section, but one important decision for each concrete instantiation is the type of entities it can process. For modeling purposes, DyKnow views the world as consisting of *objects* and *features*.

Since we are interested in dynamic worlds, a feature may change values over time. Due to the dynamic nature of the value of a feature a *fluent* is introduced to model the value of a feature. A fluent is a total function from time to value, representing the value of a feature at every time-point. Example features are the speed of a car, the distance between two cars, and the number of cars in the world.

Since the world is continuous and the sensors are imperfect the fluent of a feature will in most cases never be completely known and it has to be approximated. In DyKnow, an approximation of a fluent is represented by a *fluent stream*. A fluent stream is a totally ordered sequence of *samples*, where each sample represents an observation or an estimation of the value of the feature at a particular time-point.

To satisfy the sixth requirement of having a declarative specification of the information being generated, DyKnow introduces a formal language to describe knowledge processing applications. An application declaration describes what knowledge processes and streams exists and the constraints on them. To model the processing of a dependent knowledge process a *computational unit* is introduced. A computational unit takes one or more samples as inputs and computes zero or more samples as output. A computational unit is used by a dependent knowledge process to create a new fluent generator. A *fluent generator declaration* is used to specify the fluent generators of a knowledge process. It can either be primitive or dependent. To specify a stream a *policy* is used.

The DyKnow implementation sets up the stream processing according to an application specification and processes the streams to satisfy their policies. Using DyKnow an instance of the traffic monitoring scenario has successfully been implemented and tested [13].

VI. Related Work

There is a large body of work on hybrid architectures which integrate reactive and deliberative decision making [2–4, 18, 19]. This work has mainly focused on integrating actions on different levels of abstraction, from control laws to reactive behaviors to deliberative planning. It is often mentioned that there is a parallel hierarchy of more and more abstract information extraction processes or that the deliberative layer uses symbolic knowledge, but few are described in detail [1, 16, 17].

The rest of this section focuses on some approaches claiming to provide general support for integrating sensing and reasoning as opposed to approaches limited to particular subproblems such as symbol grounding, simultaneous localization and mapping or transforming signals to symbols.

4D/RCS is a general cognitive architecture which claims to be able to combine different knowledge representation techniques in a unified architecture [20]. It consists of a multi-layered hierarchy of computational nodes each containing sensory processing, world modeling, value judgment, behavior generation, and a knowledge database. The idea of the design is that the lowest levels have short-range and high-resolution representations of space and time appropriate for the sensor level while higher levels have long-range and low-resolution representations appropriate to deliberative services. Each level thus provides an abstract view of the previous levels. Each node may use its own knowledge representation and thereby support multiple different representation techniques. But the architecture does not, to our knowledge, provide any support for the transformation of information in one node at one abstraction level to information in another node on another abstraction level.

SCENIC [21] performs model-based behavior recognition, distributing tasks such as spatial reasoning and object recognition, classification and tracking into three processing stages: Low-level analysis, middle layer mediation and high-level interpretation. From an integration perspective the middle layer, which tries to match top-down hypotheses with bottom-up evidence and computes temporal and spatial relations, is clearly the most interesting. However, it is also quite specific to this particular task as opposed to being a general processing framework.

Gunderson and Gunderson (2006) claim to bridge the gap between sensors and symbolic levels for a cognitive system using a *Reification Engine* [8]. While other approaches mainly focus on grounding for the purpose of reasoning about the world, the authors claim that a system should

also be able to use a symbol to affect the world, citing this bidirectionality as a critical insight missing in other work on symbol grounding. The major weakness with this approach is the focus on a single step approach to connecting a symbol to sensor data, a process we believe will require several steps where the intermediate structures will be useful as well.

The CoSy Architecture Schema Toolkit (CAST) is another general cognitive architecture [9]. It consists of a collection of interconnected subarchitectures (SAs). Each SA contains a set of processing components that can be connected to sensors and effectors and a working memory which acts like a blackboard within the SA. One special SA is the *binder* which creates a high-level shared representation that relates back to low-level subsystem-specific representations [14]. It binds together content from separate information processing subsystems to provide symbols that can be used for deliberation and then action. By deliberation the authors mean processes that explicitly represent and reason about hypothetical world states. Each SA provides a *binding proxy* which contains a set of attribute-value pairs called *binding features* corresponding to the internal data in the SA. The binder will try to bind proxies together to form *binding unions* which fuse the information from several proxies to a single representation. The set of unions represent the best system wide hypothesis of the current state. A weakness is that the binder is mainly interested in finding matching *binding proxies* which are then merged into *binding unions* representing the best hypothesis about the current system state. The system provides no support for other types of refinement or fusion.

VII. Summary

As autonomous physical systems become more sophisticated and are expected to handle increasingly complex and challenging tasks and missions, there is a growing need to integrate a variety of functionalities developed in the field of artificial intelligence. A great deal of research in this field has been performed in a purely symbolic setting, where one assumes the necessary knowledge is already available in a suitable high-level representation. There is a wide gap between such representations and the noisy sensor data provided by a physical platform, a gap that must somehow be bridged in order to ground the symbols that the system reasons about in the physical environment in which the system should act.

As physical autonomous systems grow in scope and complexity, bridging the gap in an ad-hoc manner becomes impractical and inefficient. At the same time, a systematic solution has to be sufficiently flexible to accommodate a wide range of components with highly varying demands. Therefore, we began by discussing the requirements that we believe should be placed on any principled approach to bridging the gap. As the next step, we proposed a specific class of approaches, which we call stream-based knowledge processing middleware and which is appropriate for a large class of autonomous systems. This step provides a considerable amount of structure for the integration of the necessary functionalities, but still leaves certain decisions open in order to avoid unnecessarily limiting the class of systems to which it is applicable. Finally, DyKnow was presented to give an example of an existing implementation of such middleware.

References

[1] Virgil Andronache and Matthias Scheutz. APOC - a framework for complex agents. In *Proceedings of the AAAI Spring Symposium*, pages 18–25. AAAI Press, 2003.

[2] R. C. Arkin. *Behavior-Based Robotics*. MIT Press, 1998.

[3] Marc S. Atkin, Gary W. King, David L. Westbrook, Brent Heeringa, and Paul R. Cohen. Hierarchical agent control: a framework for defining agent behavior. In *Proc. AGENTS '01*, pages 425–432, 2001.

[4] P. Bonasso, J. Firby, E. Gat, D. Kortenkamp, D. Miller, and M. Slack. Experiences with an architecture for intelligent, reactive agents. *J. Experimental and Theoretical AI*, 9, April 1997.

[5] S. Coradeschi and A. Saffiotti. An introduction to the anchoring problem. *Robotics and Autonomous Systems*, 43(2-3):85–96, 2003.

[6] Patrick Doherty, Patrik Haslum, Fredrik Heintz, Torsten Merz, Per Nyblom, Tommy Persson, and Björn Wingman. A distributed architecture for autonomous unmanned aerial vehicle experimentation. In *Proc. DARS'04*, 2004.

[7] Malik Ghallab. On chronicles: Representation, on-line recognition and learning. In *Proc. KR'96*, pages 597–607, November 5–8 1996.

[8] J. P. Gunderson and L. F. Gunderson. Reification: What is it, and why should i care? In *Proceedings of Performance Metrics for Intelligent Systems Workshop*, pages 39–46, 2006.

[9] Nick Hawes, Michael Zillich, and Jeremy Wyatt. BALT & CAST: Middleware for cognitive robotics. In *Proceedings of IEEE RO-MAN 2007*, pages 998–1003, August 2007.

[10] Fredrik Heintz and Patrick Doherty. DyKnow: An approach to middleware for knowledge processing. *J. Intelligent and Fuzzy Systems*, 15(1):3–13, nov 2004.

[11] Fredrik Heintz and Patrick Doherty. Managing dynamic object structures using hypothesis generation and validation. In *Proc. Workshop on Anchoring Symbols to Sensor Data*, 2004.

[12] Fredrik Heintz and Patrick Doherty. A knowledge processing middleware framework and its relation to the JDL data fusion model. *J. Intelligent and Fuzzy Systems*, 17(4), 2006.

[13] Fredrik Heintz, Piotr Rudol, and Patrick Doherty. From images to traffic behavior – a UAV tracking and monitoring application. In *Proc. Fusion'07*, Quebec, Canada, July 2007.

[14] Henrik Jacobsson, Nick Hawes, Geert-Jan Kruijff, and Jeremy Wyatt. Crossmodal content binding in information-processing architectures. In *Proc. HRI'08*, Amsterdam, The Netherlands, March 12–15 2008.

[15] Christian Jensen and Curtis Dyreson (eds). The consensus glossary of temporal database concepts - february 1998 version. In *Temporal Databases: Research and Practice*. 1998.

[16] Kurt Konolige, Karen Myers, Enrique Ruspini, and Alessandro Saffiotti. The Saphira architecture: a design for autonomy. *J. Experimental and Theoretical AI*, 9(2–3):215–235, April 1997.

[17] D.M. Lyons and M.A. Arbib. A formal model of computation for sensory-based robotics. *Robotics and Automation, IEEE Transactions on*, 5(3):280–293, 1989.

[18] Barney Pell, Edward B. Gamble, Erann Gat, Ron Keesing, James Kurien, William Millar, Pandurang P. Nayak, Christian Plaunt, and Brian C. Williams. A hybrid procedural/deductive executive for autonomous spacecraft. In *Proc. AGENTS '98*, pages 369–376, 1998.

[19] M. Scheutz and J. Kramer. RADIC – a generic component for the integration of existing reactive and deliberative layers for autonomous robots. In *Proc. AAMAS'06*, 2006.

[20] Craig Schlenoff, Jim Albus, Elena Messina, Anthony J. Barbera, Raj Madhavan, and Stephen Balakrisky. Using 4D/RCS to address AI knowledge integration. *AI Mag.*, 27(2):71–82, 2006.

[21] Kasim Terzić, Lothar Hotz, and Bernd Neumann. Division of work during behaviour recognition – the SCENIC approach. In *Workshop on Behaviour Modelling and Interpretation, KI'07*, 2007.

Developing Intelligent Robots with CAST

Nick Hawes and Jeremy Wyatt
Intelligent Robotics Lab
School of Computer Science
University of Birmingham Birmingham, UK
{n.a.hawes, j.l.wyatt}@cs.bham.ac.uk

***Abstract*— In this paper we describe the CoSy Architecture Schema, and the software toolkit we have built to allow us to produce instantiations of this schema for intelligent robots. Although the schema does not specify a cognitive model *per se*, it constrains the space of models that can be built from it. Along with descriptions of the schema and toolkit we present the motivation behind our designs (a need to explore the design-space of information-processing architectures for intelligent systems), and a discussion of the kinds of design, implementation and information-processing models they support.**

I. INTRODUCTION

The ultimate aim of many current projects in the field of cognitive or intelligent robotics is the development of a robotic system that integrates multiple heterogeneous subsystems to demonstrate intelligent behaviour in a limited domain (home help, guided tours etc.). Typically the focus in this work is in developing state-of-the-art components and subsystems to solve a particular isolated sub-problem in this domain (e.g. recognising categories of objects, or mapping a building). In the CoSy[1] and CogX[2] projects, whilst being interested in state of the art subsystems, we are also motivated by the problems of integrating these subsystems into a single intelligent system. We wish to tackle the twin problems of designing information-processing architectures that integrate subsystems in a principled manner, and implementing these designs in a robotic system. The alternative to explicitly addressing these issues is ad-hoc theoretical and software integration that sheds no light on one of the most frequently overlooked problems in AI and robotics: understanding the trade-offs available in the design space of intelligent systems [20], [10].

The desire to tackle architectural and integration issues in a principled manner has led us to develop the CoSy Architecture Schema Toolkit (CAST) [14]. This is a software toolkit intended to support the design, implementation and exploration of information-processing architectures for intelligent robots and other systems. The design of CAST was driven by a set of requirements inspired by HRI domains and the need to explore the design space of architectures for systems for these domains. This has led to a design based around a particular computational paradigm: multiple shared working memories. Given the topic of this workshop, this paper will expand the motivation behind the design of CAST (Section II), how the computation paradigm of multiple shared working memories is implemented in our software toolkit (Section III), and how this paradigm influences the systems we build and the way we build them (Section IV).

[1]Cognitive Systems for Cognitive Assistants: http://cognitivesystems.org
[2]Cognitive Systems that Self-Understand and Self-Extend: http://cogx.eu

II. UNDERSTANDING ARCHITECTURES AND INTEGRATION

A common approach to designing and building an intelligent system to perform a particular task follows this pattern: analyse the problem to determine the sub-problems that need to be solved, develop new (or obtain existing) technologies to solve these sub-problems, put all the technologies together in a single system, then demonstrate the system performing the original task. This "look ma no hands" approach to intelligent system design has a number of problems, but we will focus on the "put all the technologies together" step. If the number of component technologies is small, and the interactions between them are strictly limited, then arbitrarily connecting components may suffice. However, once a certain level of sophistication has been reached (we would argue that once you integrate two or more sensory modalities in an intelligent robot, or would like your system to be extensible, you have reached this level), then this approach lacks the foresight necessary to develop a good system design. Instead, the initial problem analysis should cover the *requirements* for the system's information-processing architecture design (i.e. the integrating parts) in addition to the component technologies.

In the field of intelligent artifacts, the term "architecture" is still used to refer to many different, yet closely related, aspects of a system's design and implementation. Underlying all of these notions is the idea of a collection of units of functionality, information (whether implicitly or explicitly represented) and methods for bringing these together. At this level of description there is no real difference between the study of architectures in AI and software architectures in other branches of computer science. However, differences appear as we specialise this description to produce architectures that integrate various types of functionality to produce intelligent systems. Architectures for intelligent systems typically include elements such as fixed representations, reasoning mechanisms, and functional or behavioural component groupings. Once such elements are introduced, the trade-offs between different designs become important.

Such trade-offs include the costs of dividing a system up to fit into a particular architecture design, and the costs of

using a particular representation. Such trade-offs have been ignored by previous work on integrated systems, yet these factors are directly related to the efficacy of applying an architecture design to a particular problem. Our research studies architectures for integrated, intelligent systems in order to inform the designers of these systems of the trade-offs available to them.

Given that there is a huge space of possible architecture designs for intelligent systems (cf. [18], [21]) it is important both that we (as scientists concerned with designing and building such systems) understand this design space and the trade-offs it offers, and that we are able to evaluate the influence of the architectures we use in the systems we build. Although we could study architecture designs purely in theory (cf. [15]), the dynamic and complex internal and external behaviours of even the simplest robots situated in the real world means that we may have more success in studying the designs empirically. For this we require implementations of our architecture designs that allow us to separate the effects of the architecture design from the effects of the components being integrated by the design. We have argued this point elsewhere (cf. [11]), but it is worth restating as it motivates our approach to the design of middleware for intelligent robots. To avoid the uninformative, ad-hoc approach to building integrated systems (characterised above as "look ma no hands"), we must not only be able to demonstrate that our system works, but we must also be able to provide some analysis on why the system works the way it does. This type of analysis is almost always performed for the components of an integrated system, but it is rarely, if ever, performed for the architecture design used to integrate the components.

There may be many reasons why researchers do not evaluate the influence their chosen information-processing architecture has on the behaviour of their intelligent system. One reason is the confusion between the types of architectures (at various levels of abstraction) that feature in the implementation of an intelligent system. These can include very abstract information flow architectures, more concrete decompositions into subsystems and functional components, and further, more detailed, decompositions into classes and functions in software. In our experience, it is typically this latter type of architecture (i.e. the system's software architecture) that is the only architecture explicitly present in the final system implementation. Because of this, it is difficult to isolate the information-processing architecture (which we assume is typically at a higher level of abstraction than classes and functions in most systems), from the rest of the system; it is only implicitly present in the implementation. To address this problem we advocate the use of an *architecture toolkit* when building intelligent systems. Such a toolkit fixes the information-processing architecture explicitly in the implementation of the system, and keeps the architectural elements separate from the components in the system.

III. THE COSY ARCHITECTURE SCHEMA TOOLKIT

This reasoning has led us to develop the *CoSy Architecture Schema Toolkit* (CAST) [14], a software toolkit which implements the *CoSy Architecture Schema* (CAS) and thus allows us to build information-processing architectures for intelligent robots based on instantiations of this schema.

A. The Schema

As mentioned in Section II we are interested in understanding the trade-offs available in the design space of architectures for intelligent systems. We designed CAS to allow us study a small region of this space in a systematic manner. CAS is an architecture *schema*, i.e. a set of rules which can be used to design architectures. We refer to the process of designing and building an architecture from the schema as *instantiation*. By comparing instantiations of the schema we can uncover some of the trade-offs that are available within the design space defined by the schema.

CAS is based around the idea of a collection of loosely coupled *subarchitectures*, where a subarchitecture can be considered as a subsystem of the whole architecture. As shown on the left in Figure 1, each subarchitecture contains a number of processing components which share information via a *working memory*, and a control component called a *task manager*. Some processing components within an subarchitecture are *unmanaged* and some are *managed*. Unmanaged components perform relatively simple processing on data, and thus run constantly, pushing their results onto the working memory. Managed processes, by contrast, monitor the changing working memory contents, and suggest possible processing tasks using the data in the working memory. As these tasks are typically expensive, and computational power is limited, tasks are selected on the basis of current needs of the whole system. The task manager is essentially a set of rules for making such allocations. Each subarchitecture working memory is *readable* by any component in any other subarchitecture, but is *writable* only by processes within its own subarchitecture, and by a limited number of other *privileged* components. These components can post information to any other subarchitecture, allowing top-down goal creation and cross-architecture coordination.

If there are several processing goals that a subarchitecture needs to achieve, they are mediated by its task manager. This mixes top-down and data driven processing and allows goals to be handled that require coordination within one subarchitecture or across multiple subarchitectures. At one extreme the number of privileged components can be limited to one (centralised coordination), and at the other all components can be privileged (completely decentralised coordination).

In terms of design CAS falls between two traditional camps of architecture research. On one hand its shared working memories draw influence from cognitive modelling architectures such as ACT-R [1], Soar [17], and Global Workspace Theory [19]. On the other hand it grounds these influences in a distributed, concurrent system which shares properties with robotic architectures such as 3T [5].

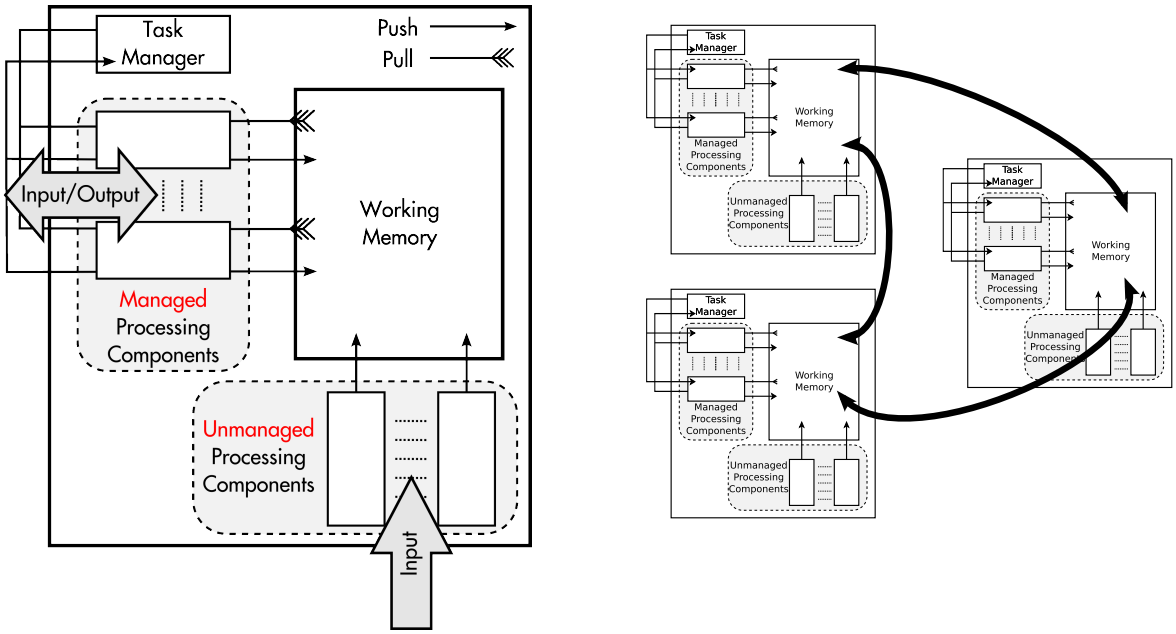


Fig. 1. Two views of the CoSy Architecture Schema. The figure on the left is the schema at the level of a single subarchitecture. The figure on the right shows how a system is built from a number of these subarchitectures. Note that all inter-subarchitecture communication occurs via working memory connections.

B. The Toolkit

This schema is implemented in our software toolkit CAST. The toolkit provides a component-based software framework with abstract classes for managed and unmanaged components, task managers and working memories. By sub-classing these components, system builders can quickly and easily create new architectures that instantiate the CAS schema. CAST provides access to functionality intended to make programming within the schema as simple as possible. Most of this functionality is based around accessing working memories.

In CAST a working memory is an associative container that maps between unique identifiers (IDs) and *working memory entries*. Each entry is an instance of a *type*, which can be considered as analogous to a class. Components can add new entries to working memory, and overwrite or delete existing entries. Components can retrieve entries from working memory using three access modes: ID access, type access and change access. For ID access the component provides a unique ID and then retrieves the entry associated with that ID. For type access the component specifies a type and retrieves all of the entries on working memory that are instances of this type. Whilst these two access modes provide the basic mechanisms for accessing the contents of working memory, they are relatively limited in their use for most processing tasks. Typically most component-level processing can be characterised by a model in which a component waits until a particular change has occurred to an entry on working memory before processing the changed entry (or a related entry). To support this processing model, components can subscribe to *change events*. Change events are generated by the working memory to describe the operations that are being performed on the entries it contains. These events contain the ID and type of the changed entry, the component that made the change, and the operation performed to create the change (i.e. whether the entry was added, overwritten or deleted).

As is necessary for any robot-centric middleware, CAST can run distributed across multiple machines. It also natively supports both C++ and Java (a requirement for our project's software which we found hard to satisfy using other robotic middleware). We currently use CORBA to provide support for the cross-language and cross-network translation, although we're considering replacing this in the future. Translation is hidden from the programmer, so that there is no difference when a component accesses a working memory written in the same language on the same machine, to when it accesses a working memory written in a different language on a remote machine. To support the exploration of the design space of CAST instantiations, architectures built with CAST can be reconfigured without changing component code or recompilation. This allows us to add and remove subarchitectures, and change the decomposition of components into subarchitectures, without changing a line of code. Finally, CAST is open source, and freely available from http://www.cs.bham.ac.uk/research/projects/cosy/cast/.

IV. THE BEHAVIOUR OF CAST

CAST allows us to design and build a wide range of intelligent systems. These systems will vary in many ways, but because they are CAST instantiations they will all share at least one feature: shared working memories. It is this feature that gives CAST systems their distinctive information-processing behaviour (a behaviour similar to distributed blackboard systems [8]). We now explore this behaviour in more detail.

A. Concurrent Refinement of Shared Information

Two of the requirements that influenced the design of CAS are the requirement that information is shared between components, and the requirement that components are active in parallel. These requirements combine in CAS to produce a model in which processing components work concurrently to build up shared representations. For example, the visual working memory in our recent intelligent system [12] contains entries representing regions of interest and objects in the scene. Once these basic structures have been created (via segmentation and tracking) other components (such as feature recognisers) can work in parallel adding information to these entries. As the data is shared, any update to an entry is available to any component that needs it as soon as the entry is changed on working memory. In this manner the results of processing by one component can by asynchronously incorporated into the processing of another as soon as the results are available.

It is not only components within a subarchitecture that are are active in parallel. Subarchitectures themselves are also concurrently active, allowing the concurrent refinement of information in one subarchitecture working memory to influence the processing of components in another. This behaviour allows us, for example, to narrow the space of hypotheses in incremental parsing [16] in parallel to extracting information about the visual scene.

Although this processing model could be implemented in a purely component-based architecture, the amount of concurrent data sharing between subsystems means that it is a very natural fit with the design of CAS.

B. Incremental Design & Development

By only allowing components in a CAST system to communicate via information shared on working memories we reduce the interdependencies between them. This allows components to be added to, or removed from, CAST instantiations without the need for the architecture be restructured (or for any code to be recompiled). This freedom allows us to adopt an incremental approach design and implementation.

Due to the subarchitecture schema design, a processing behaviour (e.g. a linguistic or visual interpretation behaviour) can be designed, implemented and tested incrementally. A designer can start with an initial set of components to perform some basic processing tasks (e.g. segmentation in the previous example of a visual system) that populate working memory. Following this, CAST allows the designer to incrementally add components to process the entries on working memory without altering the initial components (unless extensions to them are required). As long as these additional components do not introduce dependencies at the information-processing level (e.g. one component must always wait for the results of another component), then they can be added and removed as required by the task. To return to the example of the visual system, this incremental model allows us to have a basic subarchitecture that provides the 3D properties of objects. We can then add additional components (recognisers, feature extractors etc.) to this subarchitecture as required by the application domain, whilst leaving the original components untouched.

The nature of CAST has allowed us to generalise this model to whole system development. Our integrated systems feature subarchitectures developed in parallel across multiple sites. Each site follows the above approach to subarchitecture design, gradually adding components to a subarchitecture in order to add more functionality to it. The integrated system them emerges in a similar way: we start with a small number of subarchitectures containing a few components, and then add functionality in two ways. Individual subarchitectures can be extended with new components as before, but also complete subarchitectures can be added to the system (again without recompilation or restructuring) to provide new types of functionality. Because the information produced by existing subarchitectures is automatically shared on their working memories, any new subarchitecture has immediate access to the knowledge of the whole system. As with components, the restriction that communication occurs only via working memories means that additional subarchitectures can be removed without altering the initial system. In [12] we demonstrated this incremental system development model by taking an existing system for scene description and extending it with the ability to plan and perform manipulation behaviours.

We have also taken advantage of the ability to restructure a CAST system to perform a preliminary exploration of the design-space delineated by CAS. In [13] we took an existing CAS instantiation and systematically varied its ratio of components to subarchitectures. The encapsulation of the CAS-level system components into a toolkit allowed us to benchmark architectural properties of the resulting systems separately to their functional properties.

V. COMPARISONS TO EXISTING WORK

CAST can be compared to many existing software packages intended for the development of robotic systems. Although the basic unit of functionality in CAST is a processing component, it is markedly different from other component-based frameworks such as MARIE [7], ORCA [6] and YARP [9]. These frameworks provide methods for reusing components and accessing sensors and effectors, but they do not provide any *architectural structure* except in the very loosest sense (that of components and connections). It is this structure that CAST provides, along with supporting component reuse. CAST does not provide access to sensors and effectors as standard, although to date we have successfully integrated Player/Stage, various camera devices and a manipulator engine into CAST components. At this stage in the development of CAST we would ideally like to integrate it with one of these component frameworks to provide our architecture structure with access to existing devices and components.

At the other extreme of software for intelligent robotics is the software associated with cognitive modelling architectures. Such tools includes ACT-R [1] and SOAR [17]. Whilst these systems provide explicit architecture models along with

a means to realise them, they have two primary drawbacks for the kind of tasks and scientific questions we are interested in studying. First these systems provide a fixed architecture model, whilst CAST provides support for a space of possible instantiations based on a more abstract schema (allowing different instantiations to be easily created and compared). Second, it is not currently feasible to develop large integrated systems using the software provided for these architectures. This is due to restrictions on the programming languages and representations that must be adhered to when using these models. That said, researchers are now integrating cognitive models such as these into robotic systems as reasoning components, rather than using them for the architecture of the whole system (e.g. [4]).

In addition to these two extremes (tools that provide architectures and tools that provide connections) there are a small number of toolkits that have a similar aim to the work presented in this paper. MicroPsi [3] is an agent architecture and has an associated software toolkit that has been used to develop working robots. It is similar to the cognitive modelling architectures described previously in that it has a fixed, human-centric, architecture model rather than a schema, but the software support and model provided is much more suited to implementing robotic systems than other modelling projects. Our work is perhaps most similar to the agent architecture development environment ADE [2]. APOC, the schema which underlies ADE is more general than CAS. This means that a wider variety of instantiations can be created with ADE than with CAST. This is positive for system developers interested in only producing a single system, but because we are interested in understanding the effects that varying an architecture has on similar systems, we find the more limited framework of CAS and CAST provides useful restrictions on possible variations.

VI. CONCLUSIONS

In this paper we have described the CoSy Architecture Schema Toolkit (CAST) and the theoretical schema (CAS) it is based on. We discussed our motivation for developing a toolkit and also described some of the influences that the toolkit has had on the way we design and build intelligent robot systems. Although CAS does not specify a cognitive model *per se*, it constrains the space of models that can be built with it. This constrained space represents a subset of all possible architecture designs; a subset which we have found to fit naturally with the robotics problems we face on a day-to-day basis. It is worth noting that there is a related space of designs that may come from combining our schema with other middleware or architecture approaches (where this combination would typically provide a smaller, rather than larger, space of designs). We are excited to see the other contributions to this workshop and explore possible interactions between their design spaces and ours.

VII. ACKNOWLEDGEMENTS

This work was supported by the EU FP6 IST Cognitive Systems Integrated Project “CoSy” FP6-004250-IP.

REFERENCES

[1] J. R. Anderson, D. Bothell, M. D. Byrne, S. Douglass, C. Lebiere, and Y. Qin. An integrated theory of the mind. *Psychological Review*, 111(4):1036–1060, 2004.

[2] Virgil Andronache and Matthias Scheutz. An architecture development environment for virtual and robotic agents. *Int. Jour. of Art. Int. Tools*, 15(2):251–286, 2006.

[3] J. Bach. The micropsi agent architecture. In *Proc. of ICCM-5*, pages 15–20, 2003.

[4] D. Paul Benjamin, Deryle Lonsdale, and Damian Lyons. A cognitive robotics approach to comprehending human language and behaviors. In *HRI ’07: Proceedings of the ACM/IEEE international conference on Human-robot interaction*, pages 185–192, New York, NY, USA, 2007. ACM.

[5] R. Peter Bonasso, R. James Firby, Erann Gat, David Kortenkamp, David P. Miller, and Mark G. Slack. Experiences with an architecture for intelligent, reactive agents. *J. Exp. Theor. Artif. Intell.*, 9(2-3):237–256, 1997.

[6] A. Brooks, T. Kaupp, A. Makarenko, S. Williams, and A. Oreback. Towards component-based robotics. In *Intelligent Robots and Systems, 2005. (IROS 2005). 2005 IEEE/RSJ International Conference on*, pages 163–168, 2005.

[7] Carle Cote, Dominic Letourneau, Francois Michaud, Jean-Marc Valin, Yannick Brosseau, Clment Raevsky, Mathieu Lemay, and Victor Tran. Code reusability tools for programming mobile robots. In *IROS2004*, 2004.

[8] L.D. Erman, F. Hayes-Roth, V.R. Lesser, and D.R Reddy. The HEARSAY-II Speech Understanding System: Integrating Knowledge to Resolve Uncertainty. *Blackboard Systems*, pages 31–86, 1988.

[9] Paul Fitzpatrick, Giorgio Metta, and Lorenzo Natale. Towards long-lived robot genes. *Robot. Auton. Syst.*, 56(1):29–45, 2008.

[10] Nick Hawes, Aaron Sloman, and Jeremy Wyatt. Requirements & Designs: Asking Scientific Questions About Architectures. In *Proceedings of AISB ’06: Adaptation in Artificial and Biological Systems*, volume 2, pages 52–55, April 2006.

[11] Nick Hawes, Aaron Sloman, and Jeremy Wyatt. Towards an empirical exploration of design space. In *Proc. of the 2007 AAAI Workshop on Evaluating Architectures for Intelligence*, Vancouver, Canada, 2007.

[12] Nick Hawes, Aaron Sloman, Jeremy Wyatt, Michael Zillich, Henrik Jacobsson, Geert-Jan Kruijff, Michael Brenner, Gregor Berginc, and Danijel Skočaj. Towards an integrated robot with multiple cognitive functions. In *AAAI ’07*, pages 1548 – 1553, 2007.

[13] Nick Hawes and Jeremy Wyatt. Benchmarking the influence of information-processing architectures on intelligent systems. In *Proceedings of the Robotics: Science & Systems 2008 Workshop: Experimental Methodology and Benchmarking in Robotics Research*, June 2008.

[14] Nick Hawes, Michael Zillich, and Jeremy Wyatt. BALT & CAST: Middleware for cognitive robotics. In *Proceedings of IEEE RO-MAN 2007*, pages 998 – 1003, August 2007.

[15] Randolph M. Jones and Robert E. Wray. Comparative analysis of frameworks for knowledge-intensive intelligent agents. *AI Mag.*, 27(2):57–70, 2006.

[16] Geert-Jan M. Kruijff, Pierre Lison, Trevor Benjamin, Henrik Jacobsson, and Nick Hawes. Incremental, multi-level processing for comprehending situated dialogue in human-robot interaction. In *Symposium on Language and Robots*, Aveiro, Portugal, 2007.

[17] J. E. Laird, A. Newell, and P. S. Rosenbloom. Soar: An architecture for general intelligence. *Artificial Intelligence*, 33(3):1–64, 1987.

[18] Pat Langley and John E. Laird. Cognitive architectures: Research issues and challenges. Technical report, Institute for the Study of Learning and Expertise, Palo Alto, CA, 2002.

[19] Murray Shanahan and Bernard Baars. Applying global workspace theory to the frame problem. *Cognition*, 98(2):157–176, 2005.

[20] Aaron Sloman. The “semantics” of evolution: Trajectories and trade-offs in design space and niche space. In Helder Coelho, editor, *Progress in Artificial Intelligence, 6th Iberoamerican Conference on AI (IBERAMIA)*, pages 27–38. Springer, Lecture Notes in Artificial Intelligence, Lisbon, October 1998.

[21] D. Vernon, G. Metta, and G. Sandini. A survey of artificial cognitive systems: Implications for the autonomous development of mental capabilities in computational agents. *Evolutionary Computation, IEEE Transactions on*, 11(2):151–180, April 2007.

BRAHMS: Novel middleware for integrated systems computation

B. Mitchinson, T.-S. Chan, J. Chambers, M. Humphries, C. Fox, K. Gurney and T. J. Prescott

***Abstract*— Computational modellers are becoming increasingly interested in building large, eclectic, biological models. These may integrate nervous system components at various levels of description, other biological components (e.g. muscles), non-biological components (e.g. statistical discriminators or control software) and, in embodied modelling, even hardware components, all potentially with different authors. There is a need for middleware to facilitate these integrated systems. BRAHMS, a Modular Execution Framework, fills that need by defining a supervisor-process interface and an (extensible) set of process-process interfaces; authors can write to these interfaces, and processes will integrate as required. Additional benefits include reuse (never code the same model twice), cross-user readability, system-level parallelisation on multi-core or multi-node environments, cross-language integration, data logging, performance analysis, and run-stop-examine-continue execution. BRAHMS employs the nascent, and similarly general purpose, model markup language, SystemML. This will, in future, also facilitate repeatability (same answers ten years from now), transparent automatic software distribution, and interfacing with other SystemML tools.**

I. INTRODUCTION

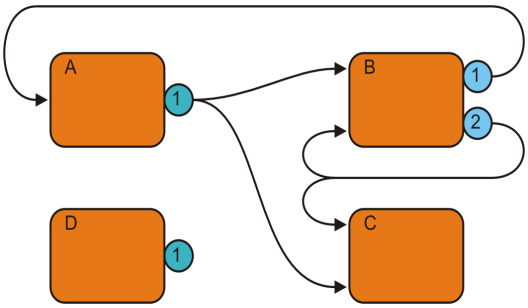


Fig. 1. Example block diagram of an integrated dynamic system built from independent processes. Rectangles represent processes, circles output ports, and arrows links between processes.

Moves are afoot in the world of computational neuroscience towards the construction of large integrated models[1], [2], [3], [4], [5], [6], [7], [8], in the spirit of Daniel Dennett's 'whole iguana'[9]. Multi-region brain models are complex, so researchers necessarily develop them as collections of sub-systems, or 'processes', which must then be 'linked' together to form a 'system' that can be computed, Fig. 1. This aside, modularity is widely considered to be a desirable trait in software design [10]. We define a Modular Execution Framework (MEF)[1] to mean any middleware that facilitates such linkage—a well-known example is Simulink[11]. Neuroscientists are wont to represent processes at different levels of abstraction (biophysical, neuronal, network, control-theoretic, etc.)[12], so it is not generally possible to compute all processes in one computation engine—rather, the task is to link *engines* together[8]. Bespoke integration serves for any single project but, as we will see, a general solution offers much more for less effort, and the startup cost is considerably outweighed by the immediate benefits.

In Section II, we run over the particular challenges and requirements for integrated computation in academic research. We go on in Section III to introduce our proposal, BRAHMS[13], beginning with an overview of its use. In Section IV, we contrast BRAHMS with existing and developing solutions and show it to be well positioned with respect to these. We report on project status in Section V and conclude in Section VI that BRAHMS already offers a solution to most of the identified challenges and will, through planned developments, meet the remainder.

Whilst BRAHMS has its roots in solving problems in and around computational neuroscience, we emphasise that it is not limited by those origins, and we expect it to be of equal interest to researchers in other fields. The problems of integration, of course, become all the more visible when crossing disciplinary boundaries.

II. CHALLENGES AND REQUIREMENTS

A. Varied Development

The primary challenge (as described above) is to integrate software processes, and the primary requirement is, therefore, to offer a middleware platform which will execute processes in concert. Processes may be developed in, on, or by different authors, labs, platforms, programming languages, human languages, programming styles and at different times. The inclusion of non-neuroscience processes generalises the problem across problem domains and technical languages. Without direct communication and refactoring, such disparate offerings will not generally be integrable. Software engineers meet such challenges by offering fixed, public, interfaces to develop against. In this context, an interface requires two facets: one between process and framework, the other between processes; these interfaces must be general (exclude no possibilities), static (or maintain backward compatibility), accessible and available in multiple programming languages on multiple platforms.

The authors are with the Adaptive Behaviour Research Group at the Department Of Psychology, The University Of Sheffield, Western Bank, Sheffield, S10 2TN, UK. Supported by European 6th Framework Grant IST 027819 ICEA and EPSRC Research Grant EP/C516303/1.

Corresponding author b.mitchinson@shef.ac.uk.

[1]We prefer 'Execution' to 'Simulation' since, in general, some processes will not be simulations.

B. Varied Deployment

High-end multiprocessing hardware is increasingly becoming available to research labs, supporting a rapid growth in the development of 'large-scale' models[14] (models with many dynamic states). At the same time, increasing focus on 'embodied modelling'[15], [16] (deployment of behavioural models on robotic hardware) is generating use cases based on low-end hardware. These two trends push the computational envelope at opposite ends, and any solution must be deployable in all these environments. A researcher may develop initially on a desktop machine, for convenience; experimental work may involve large models or parameter spaces and, thus, high-end hardware; embodied models will eventually be deployed on robots, and, particularly if there is an intellectual interest in mobile robotics *per se*, this may mean running on low-end embedded hardware. The requirements are, that a researcher should only have to develop once for such varied deployment cases, and that the middleware should be able to take advantage of the resources of high-end and desktop hardware without becoming unwieldy on low-end hardware.

C. Code Sharing

Computational researchers spend much time authoring software, and disappointingly often this work is repeated in other labs, by other researchers, or even by the same researchers, when documentation, compatible source-code and/or compatible binary code is, or becomes, unavailable. Anecdotal (and more concrete[5], [17]) evidence suggests that '...easier to rewrite it myself than try and obtain/understand/integrate their original code...' is a common story. Any solution should offer great potential for code sharing and reuse, which is to say more than that the code *could* be integrated—it must be *straightforward* to do so. This requires a (preferably, automatic) archiving/distribution mechanism, that shared code be in a form that is immediately usable (rather than having to be compiled, say), and that the solution encourages authors to document their work[18] (facilitating 'intellectual' integration).

In addition, 'background functionality' like parallelisation or data marshalling is neither trivial nor quick to author (and most researchers do not want to become software engineers), so sharing such functionality with all process developers is desirable. Therefore, as much functionality as possible should be subsumed into the middleware—'general' process code should be shared. We use the term 'supervisor' to refer to this shared code, which might, at minimum, be responsible for reading a system document, loading required processes, connecting them together, progressing them through time, collating results, and returning these to the caller.

D. Open Standards

There is more to working with systems than their execution; other possibilities include a system design GUI and an archival/retrieval tool (see also Section II-C). It is, thus, a requirement that the middleware should work with open and extensible data standards. Moreover, the needs of academic research are constantly changing and often cannot wait—the solution must be able to support these changing needs in a timely manner. In practice, therefore, it is a requirement that the solution be open source and extensible by anyone.

E. Adoption

If they are to adopt any proposal, potential users must perceive its advantages to outweigh its costs, both initially and in the long-term. The short-term requirement is met if the startup cost is sufficiently low—interfaces must be few, simple, and well documented; immediately available 'added value' will help to offset this cost. The long-term requirement is met if, in comparison with an equivalent bespoke monolithic design, overall performance does not suffer and per-process development effort is similar or less. Furthermore, an integration solution should by its nature be inclusive, so the solution must be available to all—this means affordable (preferably with no cost). Adoption will be the more willing the more freedom that is given to the developer to do things their way—this means making the interface available in multiple languages, on multiple platforms.

III. BRAHMS

A. Overview

BRAHMS represents part of our commitment to the philosophy and methodologies of neuroinformatics: developing general purpose tools that facilitate large-group working in neuroscience, and sharing and reusing resources. It is an MEF developed in-house during the course of a large-scale, multicentre project (WhiskerBot [3]) which presented many of the challenges outlined in Section II. This was a computational neuroscience project, but from the outset BRAHMS was required to integrate diverse processes (see Section III-H). The design goals of BRAHMS are performance, flexibility and extensibility. BRAHMS is open source and licensed under the GNU General Public License.

BRAHMS operates on systems, Fig. 1, progressing them through time and generating output, Fig. 2, which comprises, in large part, logs of the links between processes. Processes can be developed by dropping state initialization and update code into one of the provided templates (using a programming language that suits the developer). Systems are built from these, processes developed by other researchers, and processes provided with BRAHMS, in just a few lines of script. BRAHMS is invoked to execute these systems, taking advantage of parallel computing resources where available, and the results can easily be pulled into an analysis environment. BRAHMS is not tied to any particular interactive environment but, currently, the support offered for working in Matlab[11] is particularly strong. In time, the library of available processes will increase and additional supervisor functionality will accrue (see Section VI-B for future plans).

B. Systems

A BRAHMS system is a snapshot of a stateful dynamic system in time; that is, a collection of stateful processes and a collection of stateful links connecting them together. A system is loaded from file at invocation, and stored back to

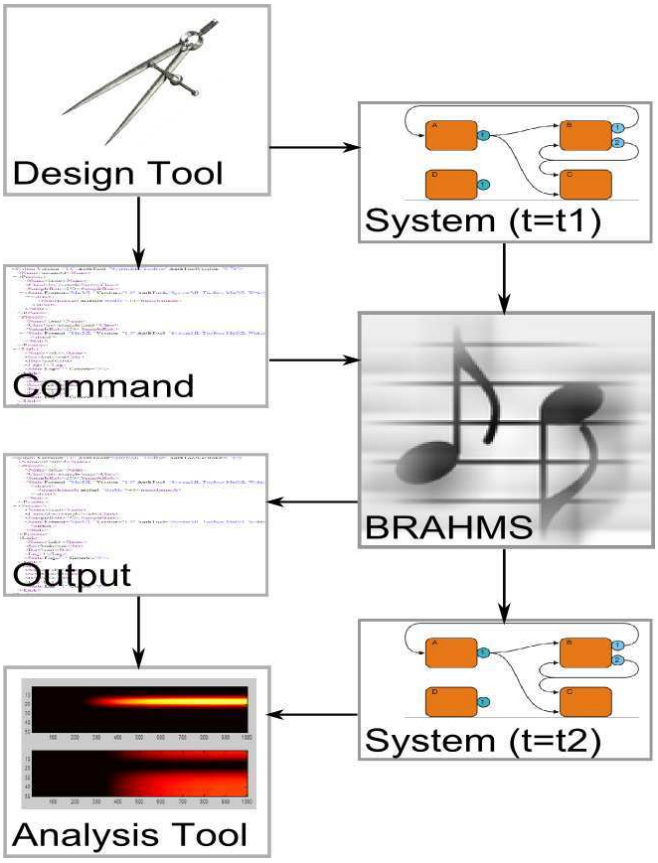


Fig. 2. Typical BRAHMS workflow: system and execution command file design, execution by BRAHMS (progression of the system through time), analysis of final system state and execution output.

file at termination (the file format used is SystemML[19], an open XML-based format for representing stateful dynamic systems). Each process is specified as a class, which indexes an extensible set of process classes each implementing some algorithm, and class-specific state data. A process may publish outputs, each of which specifies a transport protocol (periodic/aperiodic, sample rate) and a data container. Each data container is specified as a class, which indexes an extensible set of data container classes each implementing some data structure, and class-specific state data. Each link specifies a source output (implicitly, thus, a transport protocol and data container class), a destination process, and a transport delay.

The class-specific process or data container state data (termed 'StateML') is understood by design tools for and implementations of those classes (for instance, NeuroML[8] might be the StateML for some neural process classes). These class-specific data are not used by BRAHMS so the set of systems that can be represented is extensible simply by the addition of new process or data classes. Transport protocols are the responsibility of the middleware, however, so the addition of new protocols requires updating BRAHMS.

C. Interfaces

The core of BRAHMS is the supervisor-process interface—C was chosen as the language for this for its durability and interoperability with other languages. Language bindings are provided for C++ and Matlab, and additional bindings are expected to follow (Python bindings are currently under development). The interface as a whole has been designed according to modern API design principles[20], for example with an eye to minimality, extensibility, and backward compatibility. It comprises four aspects, as follows.

First, processes offer an extensible events interface through which the supervisor can invoke process operations (such as initialization, progressing through time). The remaining three aspects are callback interfaces allowing processes to invoke supervisor functionality. They are an implementation of parts of the W3C XML Document Object Model interface[21] (for manipulating StateML, amongst other things), the inter-process interface (for interacting with links), and a general interface of BRAHMS-specific functions.

In addition to the above, data container classes must offer a class-specific interface to allow manipulation of their contents by processes. Inter-process communication then proceeds as follows: the source process publishes a port during initialization, specifying the class and structure of data containers to pass over it, and this forms one end of a link; at run-time, the source process writes data into a container using the class-specific interface, and passes the container into the link; the destination process pulls the container from the other end of the link and reads the contents using the class-specific interface.

D. Modules

Process classes are implemented in modules, in one of the languages for which bindings are provided. A process module must respond to calls from the supervisor on the events interface, performing the requested operations. A simple process module might respond to only two events, first to publish an output, and second to pass data into it on each time step. Data classes are implemented in very similar modules. A data module receives a different set of events from a process module, but otherwise operation is similar. A simple data module might respond to two events, to log its current state for later retrieval, and to return its log to the supervisor. In addition, a data module must offer an interface to its content for use by processes, as mentioned above.

E. Supervisor

The BRAHMS supervisor is authored (in C++) as a standalone executable, so it does not require a virtual machine or scripting engine and is therefore resource-light. It is invoked with a command file, see Fig. 2. It then reads the system file, instantiates the system (by loading modules and passing them their state from the system file), connects processes together, then supervises the progression of the system through time, managing the transport of data through the links. At a predetermined stop time (or following cancellation by the user or by a process), it collates the state of all loaded modules (processes and data containers) and links and writes this back to a system file, also collating logs of each link specified for logging in the command file and writing these

to an output file. Most often, it is this output file of logged inter-process links that is the desired product of an execution.

Automatic parallelisation is provided at a coarse-grained (process) level. Finer-grained parallelisation can be implemented within processes, but it will generally be easier to break processes up and let BRAHMS parallelise them (models that consist of collections of similar objects, like network models, lend themselves particularly to this technique). One implementation of the supervisor, 'Solo', offers lightweight parallelisation in a shared-memory environment using multithreading. Another, 'Concerto', offers multiprocessing for parallelisation over a computing cluster, currently using either TCP/IP or MPI as the communications layer.

The data transport provided by the supervisor might comprise sharing a memory pointer (Solo), or routing a container over ethernet (Concerto), or through a stateful system file to another user, on another platform, at another time. Such transport operations are achieved without onus on the process developer. Background functionality beyond parallelisation includes inter-process data compression (Concerto), windowed data logging, distributed system performance monitoring, and a completely general 'pause & continue' execution model (allowing system snapshots at non-zero time to be coherently stored, exchanged, examined or modified).

F. Accountability

One of the dimensions of Varied Development, above, is time. This means that we should be able to integrate, today, process code that was generated many years ago. However, computing environments change, and researchers forget. BRAHMS offers accountability; that is, everything that bears on the results produced from an execution (details of each loaded library module, external library, of the run-time environment, platform, operating system, etc.) is recorded in an 'execution report'. If a repeat of the execution does not produce identical results, it is possible to identify why (thus, accountability favours repeatability by identifying sources of disagreement).

G. Software Development Kit

A BRAHMS release includes template processes and tutorial examples (corresponding to tutorials in the documentation) authored in all three currently supported languages. Thus, creating a new BRAHMS process involves little more than copying the template for the chosen programming language, and adding the content that performs the actual algorithm intended. Also included is a development version of the BRAHMS 'Standard Library', a collection of processes implementing simple operations (such as sum, product and resample) which are intended to be useful in production systems, whilst doubling as further illustrative material. This library also includes the data container class that will be most useful, 'data/numeric', which is a container for an N-dimensional array of numeric data in a comprehensive (and extensible) range of fixed-bit-width element formats.

Matlab interfaces for the command, system, and output files, and for invocation of the BRAHMS executable, are also include, such that BRAHMS can be called in Matlab using normal function call syntax. These allow the construction of systems from processes and links, and the design of the generic StateML used by the processes in the Standard Library. However, BRAHMS does not know about process state, in general, so additional tools are needed to design StateML for processes that do not use this generic StateML. Interfaces to BRAHMS from other environments are expected to follow in future.

H. Work Experience

The WhiskerBot robot is an embodied model of rat behaviour with an eclectic control model including hardware components (FPGA spiking neural simulators), neural software (leaky integrator models of Superior Colliculus and Basal Ganglia) and non-neural software (heuristic models of fixed behaviours and arithmetic/geometric modules for robot control). The model was developed on high-end desktop hardware and deployed with modification of parameters only on the low-end embedded compute platform of the robot, meeting sub-millisecond real-time constraints consistently. Large parts of the model have been adopted to contribute to the control software of the ScratchBot robot, an artefact of the ICEA project[4]. Other parts are currently in use as part of the large-scale oculomotor control model of the REVERB project[22] which deploys across a compute cluster and a separate robot-control machine. In another aspect of the ICEA project, existing WhiskerBot processes have been interfaced successfully with the Freebots[23] robot simulation environment. BRAHMS has also been chosen as the integration platform for the large European project BIOTACT[24]. Taken together, these use cases illustrate reuse, various dimensions of integration, and varied substrates of deployment.

IV. RELATED PROJECTS

We do not discuss neuroscience-only integration projects, such as NSL[25], NEOSIM and CATACOMB[26], since they do not attempt to solve the integration problem with the level of generality discussed here.

A. Simulink

Simulink[11] has a long history, and is a useful tool for learning about integrated systems. More recently, it offers multi-language support (Matlab, C, C++, Ada, Fortran) but as yet no standard support for parallelisation even within a shared-memory space, and it suffers from computational overhead. The data format is open (though not extensible, since Simulink is proprietary). In the long-term, support may improve in the technical areas where Simulink does not meet the requirements, but it is likely to remain costly, closed source and with a large resource footprint.

B. IKAROS

IKAROS[27] is a project of similar spirit to the BRAHMS project, but has rather different focus. Its positive points include the 'WebUI', which allows real-time monitoring of

system state through a browser, good documentation, and a simple developer interface. But where BRAHMS aims for to meet all the challenges listed above, IKAROS attempts to solve a much more constrained problem, albeit in a straightforward and effective way.

IKAROS is constrained to a single inter-process datatype (2D single-precision matrices), does not support dynamic creation and sizing of outputs based on connectivity, multiprocesses on a single machine only (currently), and the plugin architecture requires the whole system to be rebuilt from source when new modules are added. This last is a particular problem for accountability, since the code that executed to generate archived results is generally no longer available. There is no discussion of bindings for other languages; the IKAROS interface is authored in C++, which might become a problem in the future if the project intended to move towards dynamic loading or accountability. IKAROS also lacks some background functionality available in BRAHMS, though such features can probably, as for BRAHMS, be added without modifying the plugin codebase. In summary, we suspect that IKAROS and BRAHMS are solutions to different, if not quite orthogonal, problems.

C. Large-Scale Modeling Program and MUSIC

The International Neuroinformatics Coordinating Facility (INCF) have recently launched a program to foster infrastructure for researchers working with large-scale neural models. Attendees of the first program workshop[8] noted the large and growing set of neuron simulators available, and agreed on the importance of interoperability and component reuse. Focus was on modularity, particularly the supervisor-process interface, process-process interface, and common file format. Consequently, run-time inter-process communication was also discussed as a necessary future development to integrate computation engines into systems. They also highlighted background functionality (node allocation, communications initialisation) and marshaling of extremely large data sets.

All of these issues are addressed by our proposal. Other interoperability concerns raised in the workshop report are not applicable to BRAHMS; e.g. no application scripting is required since BRAHMS is responsible for procedure. Within the program, a communications library called MUSIC[28] is under development. The MUSIC approach leaves everything but inter-process communication to the process, in stark contrast to BRAHMS, which provides much common functionality. Each approach has its advantages—in particular, large and/or closed-source projects are more likely to suit a MUSIC interface than a BRAHMS front-end (though the latter need not be onerous). In contrast, BRAHMS allows extremely rapid development of powerful cross-platform engines, which MUSIC does not. We do not consider that BRAHMS and MUSIC will be direct competitors, and expect to offer a BRAHMS-MUSIC interface in future.

V. STATUS

BRAHMS has been public since April 2007, and is now approaching version 0.7, expected midsummer 2008. This branch will serve the ICEA, REVERB and BIOTACT projects (for three years). Interface logic is unchanged since December 2007 though minor syntax changes continue as foundations are assured for planned features. At 0.7, syntax will also freeze. Planned features will arrive in upcoming minor branches (0.8, 0.9, ...) towards version 1.0 which will end the initial development cycle. Processes authored against 0.7 will interoperate with future releases.

Solo is now fairly mature, having performed well with only minor changes for some years. Concerto still has alpha status, but is considered sufficiently mature for deployment in the REVERB project. All releases are available for Windows 32-bit, GNU/Linux 32-bit (Ubuntu) and GNU/Linux 64-bit (Debian). We anticipate offering builds for other platforms in time. Aperiodic links, pause & continue execution, and execution reports are not yet fully implemented.

VI. CONCLUSIONS AND FUTURE WORKS

A. Meeting The Challenges

BRAHMS solves the primary challenge of integration across Varied Development by specifying a common, flexible interface in multiple programming languages, against which new computational engines can be developed, and onto which existing computational engines can be imported. It meets the challenge of Varied Deployment through implementation as a lightweight standalone native executable and by allowing processes to be developed in similarly lightweight native code; a version of the supervisor is available without multiprocessing support for a further reduced footprint. BRAHMS offers extensive (and accreting) background functionality in the supervisor, meeting one aspect of the challenge of Code Sharing (a BRAHMS ‘hello world’ process written in C++ runs to about 25 lines of mostly boilerplate code, yet can distribute its complex processing across massively parallel resources).

BRAHMS supports the pragmatic challenges of sharing process code (by allowing the distribution of pre-compiled binaries rather than source code and by providing accountability) and goes some way to fostering documentation by defining a public process interface (development against a known interface self-documents to some extent, since a naïve reader knows at least some aspects of what a piece of code is intended to do). However, it does not directly address the challenge of sharing process code—see Section VI-B for details of how this will be addressed by future developments. BRAHMS employs Open Standards throughout. Adoption of the pre-release platform continues: the success of BRAHMS as the integration framework for the WhiskerBot project has led to its being chosen for three other major projects, involving varied use cases, and the early adopters have reported finding the workflow agreeable. Indications are that overall performance of systems executed using BRAHMS perform favourably in comparison with monolithic equivalent systems—quantitative metrics will follow in a later report.

New processes benefit from being built into the BRAHMS framework by taking advantage of services provided by the

system, and are constrained in their operation only by the requirements of the supervisor-process interface (i.e. a process is free to interact with the operating system, with hardware, with the user, as required). Integrating existing processes into BRAHMS can be achieved either by 'wrapping' the existing processing engine (contingent on cooperation and/or access to source code) or by meeting the communications interface of the existing software (Freebots was wrapped, for example, by authoring a BRAHMS process that communicated with it over TCP/IP).

None of the other proposals considered meet the requirements for general integration. Some discussed features are, as noted above, incomplete; however, users can begin using BRAHMS immediately and take advantage of feature additions as they become available.

B. Future Work

Above, we mentioned the SystemML file format, which is used by BRAHMS to represent systems. This open file format is a point of interface between BRAHMS and other tools. Beyond that, however, SystemML will also, in future, offer an infrastructure for the publication of processes represented in such systems. Process data in the infrastructure will include specification of parameterisation and of input/output interfaces, as well as of the algorithm itself. The infrastructure will provide archiving, distribution, version control and automatic patching (without breaking accountability) of published processes. The interplay of this infrastructure with accountability will ease the identification and removal of software bugs. This infrastructure will allow BRAHMS to meet the final challenge identified above, that of effective sharing of process code. It will also contribute to the reduction of the problem of algorithmic details becoming lost irrecoverably in undocumented optimised code.

Asked to execute a SystemML document, a naïve installation of BRAHMS will be able to obtain implementations of the specified processes through the SystemML infrastructure and execute the model without human intervention. Thus, processes published to the infrastructure will be immediately available to co-workers. Not only will co-workers be able to execute each other's models, but they will be able to use freely-available expertly-authored components in their own models.

We are committed to completing the development of Concerto, the BRAHMS Standard Library and the Python language bindings, and we expect to develop bindings for Java and Octave in future. In addition, we plan to develop interfaces for the use of BRAHMS from other interactive environments (Octave, for example). Other possibilities include a GUI system designer (operating within the SystemML space entirely, this is actually independent of BRAHMS) and the addition of a service equivalent to the IKAROS WebUI.

A future report will offer efficiency and scaling metrics, including comparison with equivalent monolithic systems.

VII. ACKNOWLEDGMENTS

The authors gratefully acknowledge the contributions of the other Adaptive Behaviour Research Group members in testing, and particularly the patience and advice of Martin Pearson as the main end user on the WhiskerBot project.

REFERENCES

[1] P. F. Dominey and M. A. Arbib, "A cortico-subcortical model for generation of spatially accurate sequential saccades", *Cereb Cortex*, 2:153-175, 1992.
[2] J. W. Brown, D. Bullock and S. Grossberg, "How laminar frontal cortex and basal ganglia circuits interact to control planned and reactive saccades", *Neural Networks*, 17:471-510, 2004.
[3] M. J. Pearson, A. G. Pipe, C. Melhuish, B. Mitchinson and T. J. Prescott, "Whiskerbot: A Robotic Active Touch System Modeled on the Rat Whisker Sensory System", *Adaptive Behavior*, 15:223-240, 2007.
[4] *ICEA*, European Union Framework 6 IST-027819, http://www.iceaproject.eu.
[5] J. M. Chambers, *Deciding where to look: A study of action selection in the oculomotor system*, PhD Thesis, The University Of Sheffield, 2007.
[6] J. G. Fleischer, J. A. Gally, G. M. Edelman and J. L. Krichmar, "Retrospective and prospective responses arising in a modeled hippocampus during maze navigation by a brain-based device", *Proc Natl Acad Sci U S A*, 104:3556-3561, 2007.
[7] B. Girard, D. Filliat, J. Meyer, A. Berthoz and A. Guillot, "Integration of Navigation and Action Selection Functionalities in a Computational Model of Cortico-Basal-Ganglia-Thalamo-Cortical Loops", *Adaptive Behaviour*, 13(2):115-130, 2005.
[8] M. Djurfeldt and A. Lansner, *Proceedings of 1st INCF Workshop on Large-scale Modeling of the Nervous System*, Stockholm, Sweden, 2006, *in* Nature Precedings, doi: 10.1038/npre.2007.262.1
[9] D. C. Dennett, "Why not the whole iguana?", *Behavioral and Brain Sciences*, 1:103–104, 1978.
[10] D. L. Parnas, "On the criteria to be used in decomposing systems into modules", *Communications of the ACM*, 15(12):1053-1058, 1972.
[11] Mathworks, *Matlab & Simulink*, http://www.mathworks.com.
[12] K. Gurney, T. J. Prescott, J. R. Wickens and P. Redgrave, "Computational models of the basal ganglia: from robots to membranes", *Trends in Neuroscience*, 27(8):453-459, 2004, doi: 10.1016/j.tins.2004.06.003
[13] B. Mitchinson and T.-S. Chan, *BRAHMS*, http://sourceforge.net/projects/brahms.
[14] M. Djurfeldt, Ö. Ekeberg and A. Lansner, "Large-scale modeling – a tool for conquering the complexity of the brain", *Frontiers in Neuroinformatics*, 2:1, 2008, doi: 10.3389/neuro.11.001.2008
[15] T. J. Prescott, F. M. Montes González, K. Gurney, M. D. Humphries and P. Redgrave, "A robot model of the basal ganglia: Behavior and intrinsic processing", *Neural Networks*, 19(1):31-61, 2006.
[16] B. Webb, *Biorobotics*, AAAI Press, 2001.
[17] R. Chavarriaga, T. Strösslin, D. Sheynikhovich and W. Gerstner, "A Computational model of parallel navigation systems in rodents", *Neuroinformatics*, 3(3):223-242, 2005.
[18] D. L. Parnas, "Software Aging", *in 16th international conference on Software engineering*, Sorrento, Italy, 2004, 279-287.
[19] B. Mitchinson and J. Chambers, *SystemML*, http://sourceforge.net/projects/systemml.
[20] J. Bloch, *How to Design a Good API and Why it Matters*, Javapolis, Antwerp, Belgium, December 12–16, 2005.
[21] Le Hors A. *et al.* (ed.), *W3C Document Object Model Core*, http://www.w3.org/TR/DOM-Level-3-Core/core.html.
[22] *REVERB*, EPSRC Research Grant EP/C516303/1, http://reverb.abrg.group.shef.ac.uk.
[23] ICEA Deliverable D27, http://www.iceaproject.eu.
[24] *BIOTACT*, European Union Framework 7 ICT-215910, http://www.biotact.org.
[25] A. Weitzenfeld, M. A. Arbib and A. Alexander, *The Neural Simulation Language: A System for Brain Modeling*, MIT Press, 2002.
[26] F. Howell, R. Cannon, N. Goddard, H. Bringmann, P. Rogister and H. Cornelis, "Linking computational neuroscience simulation tools : a pragmatic approach to component-based development", *Neurocomputing*, 52-54:289-294, 2003.
[27] C. Balkenius *et al.*, *IKAROS*, http://www.ikaros-project.org.
[28] Ö. Ekeberg, M. Djurfeldt, *MUSIC: Multi-Simulation Coordinator, Request For Comments*, http://www.incf.org, 2008.

Architecture paradigms for robotics applications

Michele Amoretti
Dipartimento di Ingegneria dell'Informazione
Università degli Studi di Parma,
43100 Parma, Italy
michele.amoretti@unipr.it

Monica Reggiani
Dipartimento di Tecnica e Gestione
dei Sistemi Industriali
Università degli Studi di Padova
36100 Vicenza, Italy
monica.reggiani@unipd.it

***Abstract*— In the latest years, several technical architecture paradigms have been proposed to support the development of distributed and concurrent systems. Object-oriented, component-based, service-oriented architectures are among the latest paradigms that have been driven the implementation of heterogeneous software products requiring complex interprocess communication and event synchronization. Despite the sharing of common objectives, the robotics community is still late in applying these research results in the development of its architectures, often relying only on basic concepts.**

In this paper we shortly illustrate these paradigms, their characteristics and successful stories within the robotic domain. We discuss benefits and tradeoffs of the different solutions with the goal of deriving some practical principles and strategies to be exploited in robotics practice. Understanding the characteristics, features, advantages and drawbacks of the different paradigms is indeed crucial for the successful design, implementation, and use of a robotic architecture.

I. INTRODUCTION

The technological development of robotics research will soon lead to the marketing of robots that can play a key role in supporting people in their everyday tasks. Pursuing a specific objective while dealing with a dynamic environment and ensuring a safe interaction with human beings, requires a complex multifunctional structure for the robot control, where heterogeneous hardware and software components interact in a coordinate manner. Additionally, the increasing number of distributed embedded computing and communication devices, available in the environment, introduces further requirements about interoperation with external systems.

The robotics community have recently proposed several architectures for robot software control [1]–[6], where monolithic development methodologies are avoided as unable to deal with the problem complexity. Despite this large number of significant proposals there is still a lack of a common, suitable solution that would allow reuse of previous efforts. The main reason for this failure is the difficulty of clearly describe and formally define a problem domain which is still unclear as the field of multifunctional robots. For the same problem, different research projects still produce different specifications for its domain. This has a huge impact on the final software architectures as it often prevents the exchange of software solutions developed by different research groups. Even if the robotics community is still not in the stage of avoiding the recreation of incompatible solutions, a plague which is common to other software research fields, it could greatly benefit from the advances and maturity reached by distributed technology research. This research field is already converging toward few technical architecture paradigms and mature implementations of these ideas are freely available in the form of software middlewares supporting complex interprocess communication, event synchronization, and data distribution. A thoughtful application of these research results in the development of robotics software architectures would, at least, alleviate the cost of re-invention of core concepts and techniques for the control of distributed devices. Nevertheless, their application to robotics research is still late, often relying only on the basic concepts of the available middlewares.

In this paper we shortly introduce three technical architecture paradigms that have been successfully exploited in several applications (Sec. II). Their characteristics and successful stories within the robotic domain will then be detailed in the following sections (III-V). We discuss benefits and tradeoffs of the different solutions with the goal of deriving some practical principles and strategies to be exploited in robotics practice. Understanding the characteristics, features, advantages and drawbacks of the different paradigms is indeed crucial for the successful design, implementation, and use of a robotic architecture.

II. ARCHITECTURE PARADIGMS

In the latest decade, the distributed computing community has witnessed a rapid evolution in the paradigms for software architectures. An increasing request for modularity, abstraction, and separation of concerns drove the development of *Distributed Object Architecture (DOA)* paradigm (Sec. III). DOA is based on the object oriented approach and is an improvement over the first attempts to provide platform independent solutions for interprocess communication such as sockets, Java RMI, etc. A following step introduced the concept of software components [7] with the objective of promoting the reuse of design and implementation efforts. The final objective of *Component Based Architecture (CBA)* paradigm (Sec. IV) is the development of components, eventually from multiple sources, that can be deployed according to customers' needs, often evolving during project lifetime. A recent trend of development of modern large-scale distributed and mobile systems is calling for a new solution able to better support an automated use of available distributed resources.

The idea of viewing software as a service is at the base of *Service-Oriented Architecture (SOA)* paradigm (Sec. V) that have been recently introduced to provide loosely coupled, highly dynamic applications.

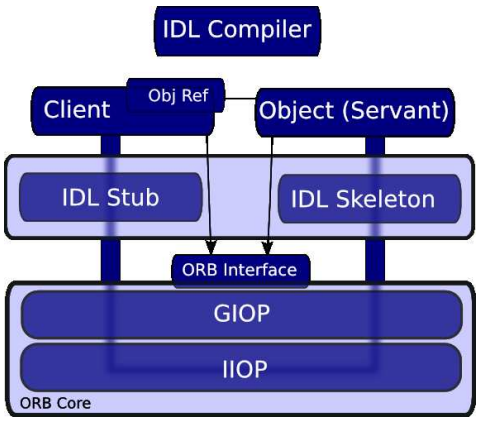
Next sections will introduce the basic characteristics of each architecture paradigm, together with few representative examples of their application in robotics domain. This lays the background required to motivate the choice among the different paradigms when a new robotics application must be developed.

III. DISTRIBUTED OBJECT ARCHITECTURE

Distributed Object Architecture (DOA) concepts are the result of the merging of object-oriented design techniques and distributed computing systems. Indeed, DOA applications are "composed of *objects*, individual units of running software that combine functionality and data" (OMG) and run on multiple computers to act as a scalable computational resource. To support the interaction between server-side objects and clients invoking them, DOA systems rely on the definition of interfaces. Each distributed object must declare the available operations so that the clients know which requests they are allowed to perform, and the DOA system knows how to marshal/unmarshal the arguments. As DOA is an evolution of object-oriented techniques, often developers identify fine-grained interfaces that need a high level of control on concurrency during multiple objects interactions.

A. DOA Standards and Middlewares

Among the several DOA proposals in the latest fifteen years, the Common Object Request Broker Architecture (CORBA) has achieved the highest level of maturity and diffusion. CORBA (`http://www.corba.org`) is a vendor-independent specification promoted by the Object Management Group (OMG) (`http://www.omg.org`) that overcomes the interoperability problem allowing smooth integration of systems built using different software technologies, programming languages, operating systems, and hardware. To ensure portability, reusability, and interoperability, CORBA architecture is based on the Object Request Broker (ORB), a fundamental component that behaves as a system bus, connecting objects operating in an arbitrary configuration (Figure 1). To achieve language independence, CORBA requires developers to express how clients will make a request using a standard and neutral language: the OMG Interface Definition Language (IDL). After the interface is defined, an IDL compiler automatically generates client stubs and server skeletons according to the chosen language and operating system. Client stub implementation produces a thin layer of software that isolates the client from the Object Request Broker, allowing distributed applications to be developed transparently from object locations. The Object Request Broker is in charge of translating client requests into language-independent requests using the Generic Inter-ORB Protocol (GIOP), of communicating with the Server through the Internet Inter-ORB Protocol (IIOP), and of translating again the request in the language chosen at the server side. Together with the Object Request Broker, the architecture proposed by OMG introduces several CORBA Services, providing capabilities that are needed by other objects in a distributed system.



1: CORBA ORB architecture

B. DOA Robotic Applications

CORBA is in wide use as a well-proved architecture for building and deploying significant robotics systems. In these sectors CORBA is growing because it can harness the increasing number of operating systems, networks, and protocols supporting real-time scheduling into an integrated solution that provides end-to-end QoS guarantees to distributed object applications.

Initial robotics work using CORBA took a simple approach to ORB technology, ignoring fundamental components such as the Naming Service for location transparency [8], or exploiting CORBA only for interoperability of previous developed components [9]. Following these experiences, other investigations used CORBA to achieve interoperability and location transparency in their applications and to exploit other useful CORBA Services [10]–[14].

Several projects for the development of robot architectures have recently based their work on CORBA. MIRO [2] is an object-oriented robot middleware freely available as open source. It supports multiple robotics platforms and common operating systems and provides a set of interfaces for communication among objects. The overall infrastructure is largely based on a client/server view build upon standards and widely used CORBA packages to simplify the integration of different robotics tasks. Humanoid control architectures have also used CORBA for the implementation of the communication layers. The large number of hardware and software components, often heterogenous, that compose a humanoid are already a distributed architecture that can benefit from distributed middlewares to simplify the software implementation [15]–[17].

The main area of applications of DOA technology is currently the development of real-time and embedded systems. Stringent requirements about computing resources and time-constraints have pushed the improvement on efficiency, scalability, and predictability of DOA middleware implementations [18]. Most of the latest application of CORBA

have been successfully applied Real-Time CORBA ORBs that allow these systems to use multithreading while controlling the amount of memory and processor resources they consume [19]–[21].

IV. COMPONENT-BASED ARCHITECTURE

Component-based architectures are based on the concept of *software component*, *i.e.* a unit of composition with contractually specified interface [22]. Following DOA approach, CBA also forces a strong separation between interface and implementation to simplify the design of large systems and promote software reuse. Nevertheless, DOA's objects are not candidates for CBA's components. While objects are tightly coupled with other objects, requiring their presence to achieve their functionality, components should be autonomous units whose purpose is well defined and understood. As a consequence, components are generally coarser-grained than objects.

Usually CBA approaches define a model that the component developers have to follow in order to allow graceful composition. Usually this model defines the creation, use, and lifecycle management of components and includes a programming model for their definition, assembly, and deployment. Interactions can follow several schemes (synchronous, asynchronous, by event, etc.) and they are usually not statically defined but can be manipulated at runtime. Additionally, to enable component composition at runtime, CBA systems should provide introspective operations to automatically discover component functionality and properties.

A. CBA Standards and Middlewares

The literature on distributed systems has proposed several implementation of the CBA concepts. The most mature and generally applicable component models include the Enterprise Java Beans (EJB) [23] of Sun Microsystems, the Object Management Groups's CORBA Component Model (CCM) [24] of OMG, and the ZeroC's Internet Communication Engine (ICE) [25]. We will limit our survey to CCM and ICE as EJB had a limited impact on the robotics domain because it is essentially tied to the Java world.

1) CORBA Component Model (CCM): The CORBA Component Model (CCM) has been proposed by Object Management Group (OMG) in CORBA 3.0 to enhance CORBA object features and have them more suitable for a component-based software development. It is a neutral open standard supporting several programming languages, operating systems, and networks, in a seamless way.

The standard extends the concept of object introducing the component model and a set of new features to simplify and automate the construction, composition, and configuration of components. Each component usually identifies a coarser unit of implementation with an interface that exposes ports for the connection with other components. Ports include facets, interfaces for synchronous method invocations, receptacles, mechanisms to declare other component interfaces required for a proper functioning, and event sources/sinks for a loose coupling among components through the asynchronous exchange of event messages.

Additionally, CCM standard specification describes the steps in the application development lifecycle. During the design, component behavior and collaboration are defined together with the required ports; the design is then implemented, requiring the definition of runtime support through component descriptors. Afterwards, component packages bundle component implementations with descriptors that are used by component assemblers connecting ports of component instances. Finally the system is deployed, preparing required resources and realizing assemblies of components.

2) Internet Communication Engine (ICE): Internet Communication Engine (ICE) is an object oriented middleware platform that has been developed by the ZeroC group as an alternative to CORBA OMG standard. ZeroC project aims at providing an efficient object-oriented middleware platform suitable for heterogeneous environments and with a full set of features to support the development of realistic distributed applications for a wide area of domains. It also aims at avoiding unnecessary complexity, for a platform easy to learn and use. While most of ICE aims are shared with other distributed middleware solutions, the attention to a code suitable for not advanced users without sacrificing the completeness of the middleware is something new and definitively not shared with most of the previously proposed solutions.

Two services included in the ZeroC middleware are of preeminent importance: ICEGrid, that implements grid computing services to discover and control remote resources, and ICEStorm, to efficiently distribute data within the architecture.

B. CBA Robotic Applications

In the latest years, an increasing number of robotic architectures have been built upon CBA principles. Indeed, the component-based approach allows to give a possible solution for several weakness on the development of software for robotics applications. A first problem is related to the great effort required to develop complete control software for robotics systems before being able to start with the implementation of research issues. The aim is to develop components for mature algorithms, sensor, and actuators that can be easily downloaded or purchased and flexibly combined. Additionally, the latest robotics applications often require distributed environments providing location transparency for easy component rearrangement on processing and bandwidth constraints. Additional details about CBA and robotics can be found in [26].

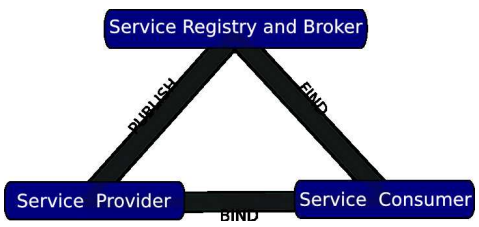
Among the several available proposals, two mature projects are RT-Middleware [27], based on OMG concepts, and ORCA [28] implemented on top of ICE. *RT-Middleware* [29] is a development framework created at the National Institute of Advanced Industrial Science and Technology (AIST). Its main goal is to simplify system integration through a methodology for the creation of Robotics Technology Component (RT-Component) and a framework

for their composition. RT-Components, built as CORBA components, consist of the following objects and interfaces: Component Object, Activity, continuously processing inputs, InPort, as input port object, OutPort, as output port object, and Command Interface. RT-Middleware supports several methods to integrate RT-Components. Together with low level integration methods directly using CORBA based system application programs or applying the composite pattern to RT-Component object structure, the framework makes available an assembly GUI tool, a script language, and XML configuration files. AIST research laboratory has also developed OpenRTM-aist, a prototype implementation based on RT-Middleware interface specification and RT-Component model that has been used to develop several testbeds such as a force controlled manipulator system [29], a service robotic system for elderly care [30] and an image recognition device [31].

ORCA is an open-source implementation framework for developing component-based robotic systems [3]. The main ORCA objective is to provide the tools for defining and developing the components that will be combined together to support the implementation of an arbitrary robotic architecture. ORCA achieves this goal through the adoption of a component-based approach using ICE for the definition of interface and communication, and with the development of tools to support the implementation of components but still keeping full access to underlying details. Through the identification of common definitions for data structures and interfaces frequently encountered in robotics, ORCA can build a repository of reusable components, libraries, and utilities [32].

V. SERVICE-ORIENTED ARCHITECTURE

Service-oriented computing defines an architectural style whose goal is to achieve loose coupling (*i.e.* minimized artificial dependencies) among interacting software entities. The key concept of this approach is the *service*, a unit of work executed by a service provider to achieve the desired results for a service consumer. Both provider and consumer are simply roles played by software entities on behalf of their owners. Therefore, service requesters (*i.e.* consumers) can be end users (provided with client tools) or other services. The interaction pattern among service providers and consumers is illustrated in figure 2.



2: Service-oriented interaction pattern

The most important achievement of SOA-based distributed environment is that shared resources (principally, applications and data) are made available on demand as independent services that can be accessed without knowledge of their underlying platform implementation.

A. SOA Standards and Middlewares

A good starting point for understanding what SOAs are is OWL-S [33], [34], a service ontology supplying a core set of markup language constructs for describing services in unambiguous, computer-interpretable form. This would allow the automatic discovery, invocation, composition, interoperation and execution monitoring of services. OWL-S is attracting a lot of interest even though it is still under development, suffering some conceptual ambiguity and lacking of concise axiomatising.

In the meanwhile, and even before the rise of OWL-S driven by the Semantic Web Community, SOAs have been mainly created and deployed using the Web Service technology. The latter aims at moving beyond the traditional middleware and framework concepts, standardizing the support of higher-level interactions, such as service flow orchestration and enterprise application integration. A number of protocols and standards define Web Services. *WSDL* (Web Services Description Language) documents describe the Web Service interface, through the identification of the supported operations and messages and their bounding to a concrete network protocol and message format. Web Service interfaces are usually listed in centralized repositories, such as *UDDI* registers, but there is still no standard protocol for distributed publication and discovery of Web Services.

The loose coupling between consumers and providers is achieved through a stateless request/reply scheme for a message-oriented interaction. Messages are typically conveyed using *SOAP*, *i.e.* HTTP with an XML serialization, but any other communication protocol could be used for message delivery. Any system supporting these standards will be able to support Web Services. Recently, the Web Services Resource Framework (WSRF) specification has been introduced to support the creation of stateful Web Services. The platform-neutral technology of Web Services has been implemented on several platforms for their development and deployment. J2EE and .NET are the most successful ones and they will be shortly introduced in the following.

1) Web Services with J2EE: Among the several J2EE competing environments, the most widely used is JBOSS [35] that provides the whole range of J2EE features. Additionally, JBOSS includes extended enterprise services including clustering, caching, and persistence as well as a J2EE certified platform for the development and deployment of enterprise Java applications, Web applications, and Portals. The open source community also provides important toolkits for Java-based development of Web Service architectures. Apache products are the most notable ones, ranging from Web Service containers to specific protocols implementations. Tomcat [36] is the servlet container belonging to the Apache suit and it is used in the official reference implementation for the Java Servlet and JavaServer Pages technologies. Axis [37] is instead the SOAP engine, *i.e.* a framework for the construction of SOAP processors such

as clients, servers, gateways, etc. It also includes a simple stand-alone server that can plug into servlet engines, such as Tomcat, extensive support for WSDL 1.1, emitter tooling that generates Java classes from WSDL, and a tool for monitoring TCP/IP packets. The latest Axis version also support advanced WS-related protocols, such as the Message Transmission Optimization Mechanism for efficient distribution of binary data among Web Services.

2) Web Services with .NET: Windows Communication Foundation (WCF) is the Microsoft platform for SOA [38]. It is a rich technology foundation that aims at building distributed service-oriented applications for the enterprise and the web. The latest version of .NET (3.0), officially launched with Windows Vista in January 2007, introduced WCF along with Windows Worflow Foundation for the support of service workflow. This marked the release of the first Microsoft web services platform for the design, implementation and deployment of services with essential plumbing for scalability, performance, security, reliable message delivery, transactions, multithreading, and asynchronous messaging.

Another important set of libraries, tools, and applications which implements the WSRF specifications is WSRF.NET, developed by the Grid Computing Groups of the University of Virginia. This free software allows easy authoring of WSRF-compliant services and clients and integrates many Microsoft technologies.

Recently, Microsoft released the Microsoft Robotics Studio (MRS) [39] a software based on .NET that provides a service-oriented architecture combining key aspects of traditional Web-based architectures with new concepts from Web Service technologies. In particular, the MRS runtime adopts the REST (REpresentational State Transfer) model as its foundation, and extends it with structured data representation and event notifications from the Web Service world. MRS supports several programming languages, including those in Microsoft Visual Studio (C# and VB.NET) as well as scripting languages such as Python.

B. SOA Robotic Applications

The adoption of SOAs in distributed robotic applications has passed through a first phase in which services were simply wrappers of existing applications, with limited exploitation of SOA protocols and tools [40]–[43]. Recently, the research community entered a second phase in which applications are (re)designed according to service-centric models, considering also advanced specifications such as OWL-S and WSRF.

In this context, a work by Ha *et al.* [44] proposes the automated integration of distributed robots, sensors and devices into ubiquitous computing environments based on semantically enriched Web Services. Their Ubiquitous Robotic Service Framework (USRF) consists of three major components: a Robotic Agent (RA), an Environmental Knowledge Repository (EKR), and Device Web Services (DWS). The RA includes a service application, a URSF Application Programming Interface (API), a plan composition module, a knowledge discovery module, a plan execution module, an OWL reasoner, and a protocol stack for Web Service execution including SOAP, XML and HTTP. To request a service from a robot, a user can input a command with a user interface for the service application. Then, the service command is encoded with vocabularies in OWL-S profile ontology and the concept ontology stored in the EKR so that the knowledge discovery module and the plan composition module can understand the user's service request. DWS are actually Web Services for ubiquitous devices including mobile robots, sensors, actuators, digital appliances, etc.

The service-oriented architecture for Web Labs proposed by Coelho *et al.* [45] is targeted to education applications. In this architecture the building blocks are services that can be recursively composed to produce other, more comprehensive, services. Lab resources (physical and logical) are modeled and implemented as services, *e.g.* a robot exports a set of services, each one performing a specific function (sensing, navigation, etc.). The concept of federation of services allows Web Labs to use resources maintained by other Web Labs located in different administrative domains. The composition logic of the experiments can be expressed in specialized languages such as BPEL (Business Process Execution Language).

Gritti *et al.* [46] explore a reactive approach to self-configuration of an ecology of robots inspired by ideas from the field of semantic Web Services, even though the resulting middleware (called PEIS-kernel) is neither based on J2EE nor .NET technologies. Their work is a clear example of how SOA principles can be decisive for solving complex distributed robotic problems. Three are the main characteristics of the proposed approach. First, there is a formal description of functionalities so they can be exported to the ecology and automatically processed. Second, a framework is available for finding exported functionalities compatible with their needs and compose them to configure at runtime a set of functionalities from different robots that can solve the current task. Finally, a mechanism for semantic interoperability allows to match functionalities from heterogeneous devices according to a unified onto logical classification.

The last application we consider is the healthcare robot platform introduced by Lee *et al.* [47] which is based on a Web Service Event-Condition-Action (WS-ECA) framework. ECA rules consists of events, notification messages from services or users, conditions, boolean expression that must be satisfied to activate devices, and actions, instructions that invoke services or generate events. The healthcare robot platform is equipped with various sensors, including ultrasonic sensors for distance measurement, IR human detection sensors, navigation sensors, etc. It can collect vital signals, such as heart rate, blood pressure, breath rate, from bio-sensors. These sensors are active publishers of context events, which can be registered by the WS-ECA engine as operators. This ECA-based approach is becoming widely used in ambient intelligence applications.

VI. CONCLUSIONS AND FUTURE WORK

In this paper we shortly reviewed three main architecture paradigms for distributed applications and their successful use within the robotics community. The survey showed that the different paradigms have different characteristics and properties that make them suitable for different distributed applications. DOA is based on the fine-grained concept of object which is suitable for lower layers where the developers need high performance even if this requires a high level of control on concurrency during multiple objects interactions. CBA and related middlewares are instead more suitable for mid-tiers where the objective is to develop autonomous components that can be exchanged and composed based on application needs. Finally, SOA is useful for the development of loosely couple architectures where the interacting entities can be accessed without previous knowledge.

Our future work will go into details in these differences to develop an in-depth discussion about influences and impacts of architecture paradigms on robotics applications, following similar investigations in other research fields [48]–[50].

REFERENCES

[1] H. Bruyninckx, "Open robot control software: the OROCOS project," in *IEEE Int. Conf. Robotics and Automation*, 2001.

[2] H. Utz, S. Sablatnog, S. Enderle, and G. Kraetzschamr, "Miro - middleware for mobile robot applications," *IEEE Transactions on Robotics and Automation*, vol. 18, no. 2, pp. 493–497, 2002.

[3] A. Makarenko, A. Brooks, and T. Kaupp, "Orca: components for robotics," in *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, 2006.

[4] R. Volpe, I. Nesnas, T. Estlin, D. Mutz, R. Petras, and H. Das, "The claraty architecture for robotic autonomy," in *IEEE Aerospace Conference*, 2001.

[5] P. Fitzpatrick, G. Metta, and L. Natale, "Towards long-lived robot genes," *Robot. Auton. Syst.*, vol. 56, no. 1, pp. 29–45, 2008.

[6] J. C. Baillie, "Urbi: Towards a universal robotic low-level programming language," in *IEEE/RSJ Int. Conf. Intelligent Robots and Systems*, 2005.

[7] C. Szyperski, "Component technology - what, where, and how?" in *25th Int. Conf. on Software Engineering*, 2003.

[8] R. L. Burchard and J. T. Feddema, "Generic robotic and motion control API based on GISC-kit technology and CORBA communications," in *IEEE Int. Conf. Robotics and Automation*, 1996.

[9] C. Paolini and M. Vuskovic, "Integration of a robotics laboratory using CORBA," in *IEEE Int. Conf. Systems, Man, and Cybernetics*, 1997.

[10] B. Dalton and K. Taylor, "Distributed robotics over the internet," *IEEE Robotics and Automation Magazine*, vol. 7, no. 2, pp. 22–27, 2000.

[11] H. Hirukawa and I. Hara, "Web-top robotics," *IEEE Robotics and Automation Magazine*, vol. 7, no. 2, pp. 40–45, 2000.

[12] S. Jia, Y. Hada, Y. Gang, and K. Takase, "Distributed telecare robtoic systems using CORBA as a communication architecture," in *IEEE Int. Conf. Robotics and Automation*, 2002.

[13] R. Siegwart, P. Blamer, C. Portal, C. Wannaz, R. Blank, and G. Caprari.

[14] T. Ortmaier, D. Reintsema, U. Seibold, U. Hagn, and G. Hirzinger, "The DLR minimally invasive robotics surgery scenario," in *Workshop Advances in Interactive Multimodal Telepresence Systems*, 2001.

[15] J. Lee, M. Jie, S. Kim, M. Jung, C. Kim, and B. You, "Balloon burster: A CORBA-based visual servoing for humanoid robot in a distributed environment," in *Annual Conference SICE*, 2007.

[16] F. Kanehiro, Y. Ishiwata, H. Saito, K. Akachi, G. Miyamori, T. Isozumi, K. Kaneko, and H. Hirukawa, "Distributed Control System of Humanoid Robots based on Real-time Ethernet," in *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, 2006.

[17] K. Takeda, Y. Nasu, G. Capi, M. Yamano, L. Barolli, and K. I. Mitobe, "A CORBA-based approach for humanoid robot control," *ndustrial Robot: An International Journal,*, vol. 28, no. 3, pp. 242–250, 2001.

[18] "The Real-time CORBA with TAO." [Online]. Available: http://www.cs.wustl.edu/~schmidt/TAO.html

[19] Y.-H. Kuo and B. A. MacDonald, "A distributed real-time software framework for robotic applications," in *IEEE Int. Conf. Robotics and Automation*, 2005.

[20] H. Song, Y. Li, F. Zhou, and L. Jia, "The Research and Application of Real Time CORBA in Software Framework for Industrial Robot," in *Int. Conf. on Integration Technology*, 2007.

[21] M. Amoretti, S. Bottazzi, S. Caselli, and M. Reggiani, "Telerobotic systems design based on real-time corba," *Journal of Robotic Systems*, vol. 22, no. 4, pp. 183–201, 2005.

[22] C. Szyperski, *Component software: beyond object-oriented programming*, ser. The Component Software Series. Addison Wesley, 2002.

[23] Sun Microsystems, "Enterprise JavaBeans Technology." [Online]. Available: http://java.sun.com/products/ejb/

[24] OMG, *CORBA Component Model 4.0 Specification*, April 2006, version 4.0. [Online]. Available: http://www.omg.org/docs/formal/06-04-01.pdf

[25] ZeroC, "Internet Communication Engine (ICE) Home Page." [Online]. Available: http://www.zeroC.com/ice.html

[26] A. Brooks, T. Kaupp, A. Makarenko, S. Williams, and A. Oreback, "Towards Component-Based Robotics," in *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, 2005.

[27] AIST, "RT-Middleware." [Online]. Available: http://www.is.aist.go.jp/rt/OpenRTM-aist/

[28] A. Brooks, T. Kaupp, and A. Makarenko, "ORCA Project." [Online]. Available: http://orca-robotics.sourceforge.net/

[29] N. Ando, T. Suehiro, K. Kitagaki, T. Kotoku, and W. Yoon, "RT-Component object model in RT-Middleware - distributed component middleware for RT (Robot Technology)," in *IEEE Int. Symp. Computational Intelligence in Robotics and Automation*, 2005.

[30] S. Jia, E. Shang, T. Abe, F. Yang, and K. Takase, "Development of service robotic system based on robot technology middleware," in *IEEE Int. Conf. Mechatronics and Automation*, 2005.

[31] A. Ikezoe, H. Nakamoto, and M. Nagas, "Development of RT-middleware for image recognition module," in *ICE-ICASE Int. Joint Conf.*, 2006.

[32] A. Makarenko, A. Brooks, and B. Upcroft, "An autonomous vehicle using Ice and Orca," *ZeroC's Connections newsletter*, no. 22, April 2007.

[33] W3C member submission, "OWL-S: Semantic Markup for Web Services." [Online]. Available: http://www.w3.org/Submission/OWL-S/

[34] D. Martin, M. Burstein, D. McDermott, S. McBraith, M. Paolucci, K. Sycara, D. L. McGuinness, E. Sirin, and N. Srinivasan, "Bringing Semantics to Web Services with OWL-S," in *World Wide Web*, 2007.

[35] "JBoss Application Server." [Online]. Available: http://www.jboss.org/products/jbossas

[36] "Apache Tomcat." [Online]. Available: http://tomcat.apache.org

[37] "Apache Axis 2." [Online]. Available: http://ws.apache.org/axis2

[38] M. L. Bustamante, "Windows Communication Foundation: Application Deployment Scenarios," MSDN homepage, May 2008.

[39] "Microsoft Robotics Studio." [Online]. Available: http://msdn.microsoft.com/robotics/

[40] M. Ghaffari, S. Narayanan, B. Sethuramasamyraja, and E. L. Hall, "Internet-based control for the intelligent unmanned ground vehicle: Bearcat cub," in *Int. Conf. on Intelligent robots and computer vision XXI : algorithms, techniques, and active vision*, 2003.

[41] V. Gilart-Iglesias, F. Macia-Perez, and J. A. Gil-Martinez-Abarca, "Industrial Machines as a Service: A Model Based on Embedded Devices and Web Services,," in *IEEE Int. Conf. on Industrial Informatics*, 2006.

[42] B. K. Kim, M. Miyazaki, K. Ohba, S. Hirai, and K. Tanie, "Web Services Based Robot Control Platform for Ubiquitous Functions," in *IEEE Int. Conf. Robotics and Automation*, 2005.

[43] K. Naruse and M. Oya, "Muscle-like Control of Entertainment Robot over Internet," in *Int. Conf. Advanced Intelligent Mechatronics*, 2005.

[44] Y.-G. Ha, J.-C. Sohn, Y.-J. Cho, and Y. Yoon, "A robotic service framework supporting automated integration of ubiquitous sensors and devices," *Information Sciences*, no. 177, pp. 657–679, 2007.

[45] P. Coelho, R. F. Sassi, E. Cardozo, E. G. Guimaraes, L. F. Faina, A. Z. Lima, and R. P. Pinto, "A web lab for mobile robotics education," in *IEEE Int. Conf. Robotics and Automation*, 2007.

[46] M. Gritti, M. Broxwall, and A. Saffiotti, "Reactive Self-Configuration of an Ecology of Robots," in *ICRA-07 Workshop on Network Robot Systems*, 2007.

[47] H. Lee, J. Park, P. Park, and D. Shin, "Development of a WS-ECA framework for a healthcare robot platform," in *Int. Conf. on Information Networking*, 2008.
[48] G. Wang and C. K. Fung, "Architecture paradigms and their influences and impacts on component-based software systems ," in *37th Hawaii Int. Conf. System Sciences*, 2004.
[49] H. Breivold and M. Larsson, "Component-based and service-oriented software engineering: Key concepts and principles," in *33rd EUROMICRO Conf. Software Engineering and Advanced Applications*, 2007.
[50] M. Jiang and A. Willey, "Architecting systems with components and services," in *IEEE Int. Conf.Information Reuse and Integration*, 2005.

Reusing software components among different control architectures with the GenoM tool

A. Mallet

***Abstract*— GenoM is a robotics component generator that has been developped and used at LAAS for the last 15 years. It is actively maintained and has been able to integrate complex robotics demonstrations on all the LAAS robots. I will present an overview of the internals of this tool, as well as the architectural principles that lead to its design. In a second part, I will present some prospective ideas on robotics component homogenization that we consider in order to foster software component reuse between research teams.**

CNRS/LAAS, Toulouse, France

Aspects of sustainable software design for complex robot platforms in multi-disciplinary research projects on embodied cognition

Martin Hülse and Mark Lee

***Abstract*— Sophisticated robot systems have become an important part in cognition research. On the one hand, autonomous robots are intended to provide a proof of concept for cognitive models. On the other hand, cognition research becomes a source of inspiration in targeting current limitations in the engineering of robust, flexible and adaptive artifacts. In this work, we discuss aspects of software development and integration for heterogeneous robotic systems in cognition research. As we will argue, one important issue is the combination of different computational paradigms within one robot system, which are rooted in the divergent approaches of engineers and scientists. This discussion lead to the introduction of a software framework aiming to overcome some well known problems of sustainable software development in robotics, but particular important for multi-disciplinary and multi-center cognition research projects. The introduced framework is based on well established standards in software engineering and therefore can be considered for a wide range of cognition research platforms and projects. Further on, we will briefly present a robotic setup where this framework is applied. It consists of a manipulator of 14 DOF (degrees of freedom) and an active vision system of 4 DOF. It is part of research activities aiming to model behavior integration and action-selection mechanisms based in large-scale neural networks.**

I. INTRODUCTION

The progress in robotic manipulation and mobile robots makes nowadays an autonomous robot platform more than an object of investigation for its own right. Miniaturization has led to platforms equipped with high dimensional sensors and actuators with many degrees of freedom able to enter the daily environment of human beings. In consequence, the focus of research and development is turning to robust, multi-modal, multi-functional and adaptive interaction of an autonomous robot system in a complex and dynamic environment. The creation of artefacts of such flexibility is still a challenge, especially if scalability is considered.

Cognition research has become one source of inspiration as well as a guidance to overcome current limitations in engineering of more complex and adaptive systems. On the other hand, cognition research projects have been utilizing robot systems as demonstrators and therefore they serve as an important proof of concept in this field. Furthermore, embodied cognition, in particular, is focused on the crucial role the body has for the development of cognitive behavior and therefore it becomes rather usual that experiments in this research involve robot systems of arbitrary complexity.

As soon as sophisticated robotic systems become part of a cognition research project one is facing a multi-disciplinary and usually also a multi-center project. Robotic engineers and scientists from different fields have to combine their approaches and know-how in order to create systems beyond the current state-of-the-art. One important area of this combination is software development. Usually multi-disciplinary research projects cannot start from scratch building up a new system. They are rather confronted with a task of integration, in which specific and very different software components are combined to one complex system. Very often these software components have been developed over years and are tightly bonded to specific constraints, such as operation system, programming language, middleware. The crucial point for robotic platforms in cognition research projects is the difference between the domain of cognition research and engineering, which is also indicated by the difference of the applied software frameworks [1]. Due to missing standards in robotics it is already difficult to extend or integrate robot systems without considering high-level cognitive models. Hence the integration of software developed in the domain of cognitive science and robotics becomes rather a challenge of its own.

The objective of this paper is to outline crucial aspects that become relevant in software engineering of robotic systems in cognition research projects. Based on our experience and recent reviews on software development and integration in robotics we have developed a framework for medium and large projects aiming for complex experimental robotic platforms for cognitive models. This framework is purely conceptual, based on design patterns and standards in software engineering, and can therefore be applied to any hardware and software environment. Furthermore, we will explain some elements of this framework in detail, based on examples of an ongoing project basically involving a manipulator equipped with a three-finger system and a stereo-vision system. This experimental platform is used for the development of large-scale neural models coordinating reaching and grasping tasks.

This paper is organized as follows. The next section introduces the key aspect influencing the software integration in cognition research projects. After this, the following two sections give an overview of the-state-of-the-art in software development / integration in robotics and outline our proposal of a framework considering aspect of robotics in cognition research. This is followed by a section which gives a concrete example of this framework leading to the concluding section of this work.

Dept. Computer Science, Aberystwyth University, Penglais, SY23 3DB, Wales, UK {msh,mhl}@aber.ac.uk

II. KEY ISSUES OF SOFTWARE DEVELOPMENT IN ROBOTICS

The problem of sustainable software design has been an important issues in software engineering in general and therefore is widely and continuously discussed in many fields and domains. Guiding issues like modular, interoperable and reusable software have led to the promotion of object-oriented design principles, design patterns as well as agile and test-driven software design methods. These issues are relevant in robotics indeed, but shall not be discussed here. Nevertheless, the following collection outlines specific aspects of software design crucial for robotic related multi-disciplinary research projects.

A. Prototypes and multi-components

Sophisticated robot systems are build up of different components. Sensors, actuators and mechanics supposed to establish a coherent robot system are very often products of different manufactures. Sometimes these components have even the character of a prototype, i.e. software and hardware are not sufficiently tested and might lack in specific functionalities and robustness. Furthermore, it is not unusual that the delivered driver software of hardware devices is very rudimentary, though one might expect software providing already solved and well known standard applications.

In consequence, for robotic components the first step is the development of a software which provides robust and general functionality and a proper error handling mechanism. This includes also sufficient test cases, which support a robust and smooth exchange of system components, if firmware and / or hardware devices must be upgraded or exchanged.

B. Different representations and levels of abstraction

In research laboratories it is common that one device, i.e. a whole robot system or a specific component, is used for experiments in different domains. This might be necessary because experiments in a project must be conducted on a lower level of functionality in oder to decide future design issues. On the other hand, a research laboratory might be involved in other projects, currently or in future, and so it is essential that specific components can efficiently be used in very different experiments.

Therefore it becomes important for the software design to provided different levels of abstraction and data representations and of course this should take as little effort as possible. This refers to the need that the core functionality of robotic devices can be used independently, and that the exchange and extension of system services with respect to hardware and software must be provided.

C. Distribution of computational resources

Robotic system components might only work in a specific software environment. Some devices might also run on specific hardware, such as FPGAs. It is also usual that computational expensive processes have to be distributed over different computers in order to guarantee real-time constraints. Recent experiments in neuroscience also show that clusters might be necessary to simulate large-scale neural models driving a robot platform [2]. Hence, nowadays robust, transparent and reliable interprocess communication is a need for almost any nontrivial autonomous robot system.

D. Distributed teams

Where cognitive science and robotics meet it is very likely that developers of specific system components are geographically distributed. The exchange of source codes and software libraries (sometimes even only pre-compiled) via suitable software repositories becomes only one major issue to consider in this process. Due to the division of knowledge and competence within a project it also very likely that software integration between the different partners is rather vertical instead of horizontal.

Horizontal integration means that two or more project partners deliver software which is horizontally organized within the overall software architecture. For instance, one team delivers the hardware and software of sensor type *A*, while another team is doing so for sensor type *B* and another team is responsible for an actuator *C*. All three software components can be developed independently.

A vertical integration starts if a fourth party develops applications *X* on top *A*, *B* and *C*, taking data from *A* and *B* and generating data feed into *C*. The success of this type of integration depends on very carefully defined and implemented interfaces. Since formal interface definition languages are purely syntactic and cannot cover any semantic information, this process must involve an understanding of the constraints and needs of each part in a reasonable depth. This usually requires time, rather days than hours.

E. Simulator

Almost every complex robotic project sooner or later requires the use of a simulator, especially if an autonomous robot system is intended as a test platform for learning or other forms of self-organized mechanism of adaptation. Simulations are an important tool to provide a proof of concept for new methods and in order to tune important system parameters in advance. However, it only makes sense to use simulators if the control software under investigation generates the same qualitative behavior in simulation as on the real robot. It is also important that the same control software can directly be used for both, simulator and real platform, without any parameter changes or even refactoring.

F. Integration of different paradigms

Robotic related cognition research projects have to pay particular attention to the coupling between high-level cognitive models and hardware specific software. Cognitive models are grounded in specific paradigms of computation and knowledge representation. Consequently, this leads to model-implementations based on declarative or functional computer languages or even simulations of neural networks. In contrast hardware-close software is usually developed in procedural computer languages strictly following this paradigm.

The problem with different paradigms is that sometimes specific constraints present either in the higher-level model or

in the lower-level software cannot directly be represented in the other domain. Hence, these constraints cannot be handled at all and therefore cut the overall system performance. Examples relevant in almost any systems combining robot hardware and cognitive models are real-time constraints and the different time-scales that specific system components are operating on.

In consequence a lot of effort must be put into developing efficient pre- and post-processing, scheduling and error-handling for bridging robot hardware and cognitive models.

G. Flexibility

The aspect of different paradigms leads to another problem, also described in [1]. An engineer creates systems, whose component functions are most efficient when they meet a detailed set of specifications exactly. The consequence is high performance for a very specific task. But as soon as the application domain is extended or becomes more general a decline of performance must be expected.

On the other hand, higher-level robotic applications, and cognitive scientists are no exception, they develop their models, applications and experiments in a language grounded in an ontology based on general principles. Hence, they expect reasonable and scalable performance for general domains and problem spaces.

The aspect of interface definition and description was already described in section II-D for geographically distributed teams. In this context, interface definition and implementation become crucial because of the multi-disciplinary character of cognitive science and engineering, thus, the different approaches and the divergent expectations of specific and general system performance.

In fact, one has to accept the inevitability of different understandings between cognitive scientists and engineers about the needs and the relevance of specific elements of the targeted models and tasks. This discrepancy is often overseen at the project-start but will emerge as soon as lower and higher level implementations meet. As a matter of fact, the re-definition of interfaces, frameworks or even experiments will be the consequence. In our experience such re-definitions will happen several times in larger projects and always go hand in hand with refactoring of certain extents. It is therefore, important to be aware of this problem, and on the other hand to provide a software engineering framework which allows, with reasonable effort, the alteration of the interfaces and the corresponding implementations on both sides: robotic hardware functionality and high-level cognitive models.

III. STATE OF THE ART

Robotics community is aware of the first five problems issued in II-A – II-E, but very little attention is focused on the problem of different paradigms. Nevertheless, standards providing robust and flexible solutions for interoperable, reusable robotic software does not exist yet. Although this lack of standards is recognized by many researchers, the most common solution to overcome this problem is to develop a new software. Mostly such implementations are claimed to be more general, but indeed, are addressing only specific needs and even more crucial the software is even immature. Consequently, it is not used by other labs and therefore is far away from providing a base for any standard.

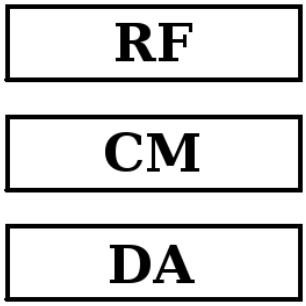


Fig. 1. The three software architecture layers of system design in robotics. *DA*, driver and algorithms, *CM* interprocess communication layer and *RF* robotic control framework, see also [3], [4] and [10].

Noticeable is the effort in many robotic projects developing middleware for a framework of handling distributed robotic systems. However, interprocess communication is an important aspect but not the only one for autonomous robots. Recent reviews [3], [4] show that in robot systems basically a 3-layered software architecture must be considered (see Fig. 1). In the lower level *DA* software driver and algorithm implementations are located. The middle level *CM* provides interprocess communication. The top level *RF* is the place where the actual robotic control frameworks are developed. It is this top layer where robotic projects implement their strategies and models, generating intelligent behavior. Some research projects claim that this level should provide a declarative programming framework [3], because they think it is the best way to implement intelligent behavior. However, other projects would probably disagree introducing a different framework for this level, which matches best with their paradigm of computational intelligence.

Being aware of the subjectivity and biased view on the top level, current activities in developing general robotic programming frameworks are primarily focused on the two lower levels: *DA* and *CM*. *Player* [5], for instance, delivers a framework, where *DA* and *CM* are interwoven [4]. The *YARP* software [6] actually provides only a framework for the *CM* layer. The developers of *MIRO* [7] had similar intentions. However, they have built *MIRO* as an extension of *CORBA* in order to make this powerful middleware standard easier to handle and faster to learn. *ROCI* [8] is based on the philosophy that complex robot behavior is achieved by "wiring" irreducible modules. In consequence, this software provides the design of modules acting in a decentralized manner. Therefore, in *ROCI* all three layers collapse into one network of interacting primitive modules.

Another strategy called *MARIE* [9] tries to support the reuse of existing programming environments and their connections through a common middleware framework. Being aware about the missing standards in interprocess commu-

nication *MARIE* provides basically a set of design patterns able to integrate communication protocols present in systems composed of heterogeneous hardware components. However, the whole design is based on middleware.

Once again, these examples represent the focus on the *CM* level, which shows that system design is almost completely seen as a problem of reliable communication between the components. However, reliable and transparent communication has always an offset. This offset is crucial if many relative primitive interacting components have to cope with real-time constraints. What we want to emphasize is, if a framework based on interprocess communication is applied for system integration, it follows that, the lower the level of system functions the more the reduction of system performance due to the offset of communication.

It is this observation, that led us to the formulation of a framework which tries to keep the middle layer *CM* as "thin and high level" as possible. However, interprocess communication provided in *CM* is an essential part in order to connect high-level cognitive models and robotic hardware. But in using it very sparely one can apply computational expensive but standardized and mature middleware solutions. In doing so, one has a wide coverage of different software environments and on the same time one can handle many effective real-time constraints on lower level functions without the involvement of computationally expensive middleware. This reduces also the effort needed for refactoring the interfaces between high-level cognitive models and robot hardware. However, the problem of modular, interoperable and reusable software design in the basic layer *DA* must still be addressed explicitly. We have done this by the usage of specific design patterns, which will be explained in the next section.

IV. GENERAL FRAMEWORK

The general framework of our software architecture is based on the the three-layered architecture as show in Fig. 1. However, in the need for the support of a sustainable software design in *DA* we have divided this lower level into two levels: *API* and *MVC* (see Fig. 2).

The lower *API* provides simple and almost purely hardware related application interfaces. These interfaces provide common and general functionality for specific hardware devices, such as cameras, laser scanners, actuators. Although these implementations will be usually very simple and straight forward, they shall already make use of an object-oriented design. Also important is the testability of each component and the support for other software developers through documentations and basic example applications. It is also necessary that the components in the *API* layer can independently be used and developed. This ensures the smooth integration or update of new hardware and firmware.

On top of these APIs we only develop new system functions based on the model-view-control design pattern [11]. This design pattern supports a complete separation of hardware from applications and the first level of abstraction. While the model element provides all the hardware functionality the view and control processes can independently

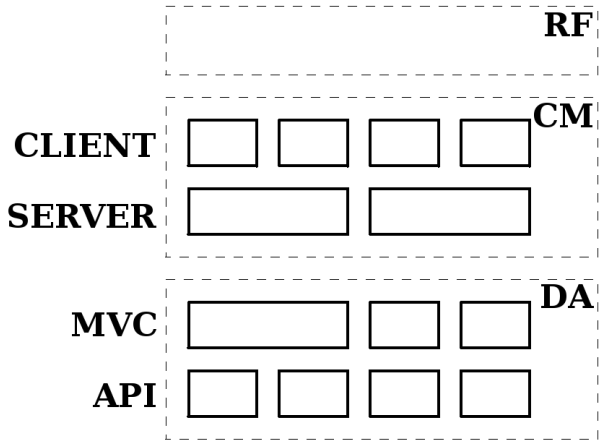


Fig. 2. The proposed software architecture framework in relation to the 3-layered variation in Fig. 1.

and parallel interacting with model. Different views can provide different representations of the current data even taking into account temporal aspects. Control elements can either monitor and maintain defined constraints or instantiate more sophisticated control schemas. It is also possible to combine several *API* components within one model.

As we have argued above, the middle level *CM* primarily connects the cognitive model implementations in *RF* with the robot hardware and related services. Since the software environment for cognitive models and robotic hardware is probably very different, it is recommended to use standard middleware solutions in order to cover as much diversity as possible. And to our knowledge, these standards in the domain of distributed systems will all support client-server frameworks. Hence, robotic functionalities and services can be provided by one or more server applications, while clients are responsible to request and deliver data needed and generated by the cognitive model running in the *RF* layer. Notice, the usage of standard middleware solutions also provides the distribution of lower robot functions, because different servers can run on different machines.

V. ROBOTIC SETUP FOR THE REVERSE ENGINEERING OF THE VERTEBRATE BRAIN

The above introduced framework is applied in a project, called REVERB [14], in which behavior integration and action-selection mechanisms are modelled based on biologically inspired large-scale neural networks. These models are tested and developed on a robot platform basically consisting of a 14 DOF (degrees of freedom) manipulator and a vision system. The manipulator integrates a 7 DOF *Lightweight arm LWA3* and a 7 DOF *Dextrous Hand SDH*. Both devices are manufactured by SCHUNK GmbH & Co. KG [13]. The vision system is based on a 4 DOF pan-tilt-verge platform equipped with two firewire cameras and a *SCAMP* vision system [12]. The unique feature of the *SCAMP* system is basically the pixel-per processor vision chip based on analog technology. This allows the execution

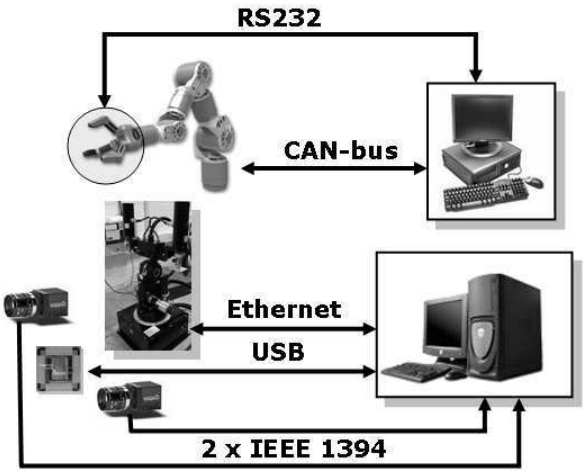


Fig. 3. Robot hardware and corresponding distribution over two PCs. The overall experimental platform consists of a manipulator of 14 DOF and vision system involving a pan-tilt-verge system, two firewire cameras and a SCAMP vision system [12].

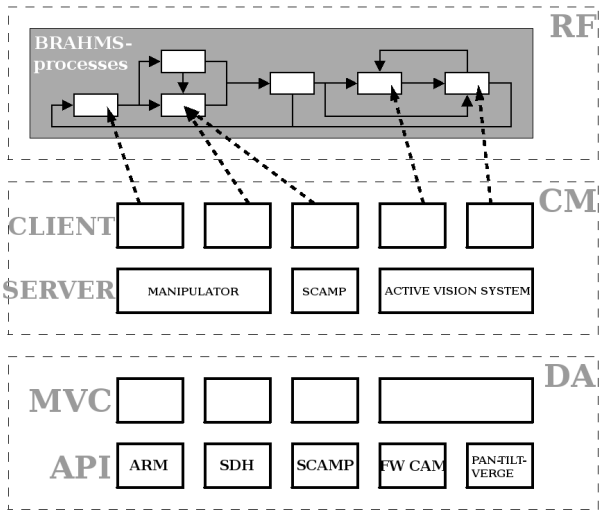


Fig. 4. Software architecture, see text for explanation.

of computational expensive image processing algorithms in real-time.

Almost typical for the integration of different devices, each is based on a different communication channel, such as CAN-bus, serial, USB, firewire and ethernet. Currently, the two main hardware components are even connected to different computers, (see Fig. 3).

A. Software architecture

Our software architecture has five independent components in the API layer; each for every hardware device: camera, pan-tilt-verge system, *LWA3*, *SDH* and *SCAMP*. On top the *MVC* layer integrate some but not all of these components. *LWA3*, *SDH* and *SCAMP* have still their own model-view-control implementation, while pan-tilt-verge and cameras are integrated in one pattern (Fig. 4). This does not mean that in the future no other additional patterns will summarize other components. This depends on the development in the project.

The applications in the *MVC* layer are wrapped by CORBA server implementations providing an interface for interprocess communication and distribution. CORBA-client implementations in arbitrary software environments are now able to access these hardware components and the services provided in the *MVC*-level. Due to usage of CORBA the interfaces must be written in IDL (interface description language). This provides, at least on the syntactical level, coherent interface definitions between low and high level functionality.

Actually CORBA-clients are part of the processes which establish the overall target of this software organization, that is the cognitive model implementation. As we have mentioned, the cognitive model in this process is implemented by large-scale artificial neural networks. The software used to simulate these networks is called *BRAHMS* [15]. Among many features, with *BRAHMS* one is able to link different processes in an arbitrary manner. In such a way, different layers of artificial neural networks can be connected, even recurrent. The processes can be implemented within a general C++ environment and the connectivity with other processes is defined in a XML-based language. It is also possible to simulate the system in a MATLAB environment.

Due to its general character, CORBA-clients can straightforward be instantiated in *BRAHMS*-process. In such a way, a distributed robot systems becomes part of a large-scale neural model, which is simulated in *BRAHMS* and might itself be executed on a cluster.

B. The usage of design patterns

The benefit of model-view-control based implementations might be best briefly demonstrated by the following two examples. We have outlined above that reusability of software involves the alterations of data representations and the level of abstractions. For visual information this means, that image data might be applied to different filters or feature detection processes. Hence, we have used the views in the MVC-design pattern in order to deliver different filters. The instantiations of the view processes operate independently and parallel. On one side, this supports the exploitation of multi-processor system, but more important, the implemented filters can be applied to any future instantiations of the corresponding design pattern. Therefore, a set of independently used functions can be generated which is totally separated from the underlying hardware.

As only one example for the *LWA3* 7 DOF arm system we have implemented a simple arm coordination task based on two independently working control processes. The arm coordination task is simply: while arm is moving, the orientation of the last segment, the hand segment, shall remain the same.

The corresponding MVC-pattern is initiated with only two control process. The first is responsible for the global orientation of the arm, while task for the second process is

to keep the orientation of the SDH hand in space constant. The bottom-line of this example is that the first process is actively changing the global configuration of the arm, while the second is passively adjusting the remaining DOF, which, in this case, maintains the orientation of the hand. Both control processes are operating in parallel on the same data. This avoids inconsistencies and makes the overall control much easier.

C. Switch between simulator and real robot system

The usage of CORBA supports also the integration of simulators. As we have seen it for the robot hardware a CORBA server can also be based on a robot simulator. If both server implementations are based on the same interface definition given in IDL, it makes actually no difference for a client to which server it is talking. Hence, without any changes in the client, it can communicate either with the simulator or with the real robot. This type of integration is successfully applied, for instance, for the mobile robot platform KURT3D and the corresponding simulation MACSim [16], [17].

D. Summary

The brief introduction of our robotic platform already outlines the importance of two key aspects in our software architecture: *MVC* design patterns and CORBA as widely supported middleware standard. The *MVC* patterns guarantee the strict separation of APIs and application layer right from the beginning. This supports the independent, modular, test-driven, and scalable software design of robotic components. We have also elucidated, how *MVC* patterns can simplify the control and provide different data representations. Complex, multi-modal and computationally expensive algorithms can already be implemented in the *MVC*-level without the usage of middleware.

The usage of *MVC* allows the integration of the *CM* layer on a much higher level of abstraction, which can lead to the reduction of interprocess communication. Therefore, powerful and computationally expensive middleware standards, like CORBA, can be applied without violating real-time constraints in the overall system. As we see in our example CORBA supports as wide range of software environments, which enables us to couple our robot hardware with a MATLAB framework. Further on, the IDL used in CORBA provides robust interface definitions between different developer teams and totally different data sources, such as a simulator. It is this last issue, which enables us to run a cognitive model either on a real robot or a simulator without any changes.

VI. CONCLUSION

Focused on current standards in software engineering we have introduced a software architecture particularly developed for robotic systems made of heterogeneous hardware devices and components. We have outlined how model-view-control design patterns and CORBA, as the leading middleware standard, can provide a sustainable software development for different levels of abstraction. As we have argued, this supports the integration of different computational paradigms. The last aspect makes our framework particularly interesting for robotics in cognition research, where engineers and scientists from different fields must integrate their different ways of system design and modelling.

VII. ACKNOWLEDGMENTS

The authors gratefully acknowledge the support of EPSRC through grant EP/C516303/1.

REFERENCES

[1] A. Farinelli, G. Grisetti and L. Iocchi, ”Design and implementation of modular software for programming mobile robots” *Int. J. of Advanced Robotic Systems* vol. 3, 2006, pp. 37-42

[2] G.M. Edelman, ”Learning in and from Brain-Based Devices” *Science*, vol. 318(5853), November, 2007, pp. 1103-1105

[3] I.A.D. Nesnas, R. Simmons, D. Gaines, C. Kunz, A. Diaz-Calderon, T. Estlin, R. Madison, J. Guineau, M. McHenry, I-H. S. and D. Apfelbaum. ”CLARAty: Challenges and Steps Toward Reusable Robotic Software” *Int. J. of Advanced Robotic Systems* vol. 3, 2006, pp. 23-30

[4] A. Makarenko, A. Brooks and T. Kaupp, ”On the Benefits of Making Robotic Software Frameworks Thin” *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS'07)*, San Diego CA, USA, 2007.

[5] T.H.J. Collett, B.A. MacDonald and B.P. Gerkey, ”Player 2.0: Toward a Practical Robot Programming Framework” *Proc. of the Australasian Conf. on Robotics and Automation (ACRA 2005)*, Sydney, Australia, December 2005.

[6] G. Metta, P. Fitzpatrick and L. Natale, ”YARP: Yet Another Robot Platform” *Int. J. of Advanced Robotic Systems* vol. 3, 2006, pp. 43-48.

[7] G.K. Kraetzschmar, H. Utz, S. Sablatnög, S. Enderle and G. Palm ”Miro - Middleware for Cooperative Robotics” *RoboCup 2001: Robot Soccer World Cup V*, LNCS 2377, Springer, pp. 411-416.

[8] A. Farinelli, G. Grisetti and L. Iocchi, ”Design and implementation of modular software for programming mobile robots” *Int. J. of Advanced Robotic Systems* vol. 3, 2006, pp. 37-42.

[9] C. Cote, D. Letourneau, F. Michaud and Y. Brosseau, ”Robotics System Integration Frameworks : MARIE's Approach to Software Development and Integration” Springer Tracts in Advanced Robotics : Software Engineering for Experimental Robotics, Springer, vol. 30, March 2007.

[10] R.P. Bonasso, D. Kortenkamp, D.P. Miller and M.G. Slack, ”Experiences with an Architecture for Intelligent Reactive Agents” *Proc. of the Int. Joint Conf. on Artificial Intelligence*, 1995.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlisside ”Design patterns: Elements of Reusable Object-Oriented Software” *Addidon-Weslay*, USA, 1995.

[12] D. R. W. Barr, P. Dudek, J. Chambers and K. Gurney, ”Implementation of Multi-layer Leaky Integrator Networks on a Cellular Processor Array” *Int. Joint Conf. on Neural Networks*, IJCNN 2007, USA, 2007.

[13] SCHUNK GmbH & Co. KG, *http://www.schunk.com*, 2007.

[14] REVERB: Reverse Engineering the VERtebrate Brain, *http://www.abrg.group.shef.ac.uk/projects/reverb/public/* EPSRC Research Grant no. EP/C516303/1, UK, 2007.

[15] BRAHMS, *http://sourceforge.net/projects/abrg-brahms*, 2007.

[16] Rome, E.; Paletta, L.; Sahin, E.; Dorffner, G.; Hertzberg, J.; Breithaupt, R.; Fritz, G.; Irran, J.; Kintzler, F.; Lörken, C.; May, S.; Ugur, E. The MACS project: An approach to affordance-inspired robot control. to appear in: Towards Affordance-based Robot Control, Proceedings of Dagstuhl Seminar 06231, Springer, LNAI 4760, Rome, E., Hertzberg, J. and Dorffner, G. (eds.), 2007

[17] E. Ugur, M.R. Dogar, M. Cakmak and E. Sahin, ”The learning and use of traversability affordance using range images on a mobile robot”, *Proc. of IEEE Int. Conf. on Robotics and Automation*, ICRA07, 2007.

A modular architecture for the integration of high and low level cognitive systems of autonomous robots

Michael Spranger, Christian Thiele, and Manfred Hild

Abstract— **This paper presents an actively developed and used software framework that integrates different computational paradigms to solve cognitive tasks of different levels. The system has been employed to empower research on very different platforms ranging from simple two-wheeled structures with only a few cheap sensors, to complex two-legged humanoid robots, with many actuators, degrees of freedom and sensors. It is flexible and adjustable enough to be used in part or as a whole, to target different research domains projects and questions, including Evolutionary Robotics, RoboCup and Artificial Language Evolution on Autonomous Robots (ALEAR). In contrast to many other frameworks, the system is such that researchers can quickly adjust the architecture to different problems and platforms, while allowing maximum reuse of components and abstractions, separation of concerns and extensibility.**

I. INTRODUCTION

Current cognitive robotics research is a wide, interdisciplinary field, which sees contributions from such diverse fields as psychology, biology, neuroscience, linguistics, computer science and artificial intelligence. The reason for this wide spread influence of diverse fields lies at the heart of building complete artificial systems, which resemble and achieve different levels of intelligence, autonomy and developmental capabilities. The manifold contributions of different fields pose not only great opportunities but also great challenges. Specifically when trying to understand and model rich phenomena such as for example language in a full systems approach, the involved subsystems easily get complex and hard to manage. However, the research in language evolution has greatly benefitted from groups trying to build models capable of dealing with noisy real visual data, motor control of real robots and populations of real robots [1], [2], [3], [4]. Often the solutions found in such full system scenarios differ significantly from simulated approaches [5] leading to more robust, plausible and scalable systems [6]. Even more, some people (especially in robotics) vigorously stress the role of embodiment and sensorimotor integration for intelligence and intelligence research [7], [8].

The system presented in this paper is used on multiple robots in experiments targeting very different research domains. One such domain is sensorimotor control, which is explored in the current framework using a neural network dynamical systems approach and artificial evolution. The framework provides a layer of abstraction which has been built to allow for research on different hardware platforms ranging from simple two-wheeled robots to two-legged humanoid robots with many degrees of freedom. A second layer is concerned with higher level perception, world modeling, reasoning and planning. This second layer is used to explore visual processing, cognitive environment mapping and deliberative and reflexive reasoning tasks. The concrete task at hand is governed by the research domain targeted by a particular researcher.

Our group is involved in two big research topics: RoboCup and artificial language evolution. The presented architecture is used in both scenarios. RoboCup [9] is an international research effort to push research in autonomous systems by introducing a dynamic scenario in which robots have to play soccer in teams against each other. The domain requires robots to perceive their environment, model it and given appropriate representations ultimately requires balancing deliberate and reactive behavior decisions. The second research domain concerns artificial language evolution on autonomous robots. This domain focusses on experiments in which autonomous humanoid robots self-organise rich conceptual frameworks and communication systems with similar features as those found in human languages. Language and cognition are seen as complex adaptive system shaped in interaction with the environment and in interaction with other population members. That is, language and cognition are grounded perceptually in the world and ones own body, but also socially through repeated communicative interactions in a community of agents. Both research domains are targeted at building complete artificial systems. The whole chain of information processing is taken into account to solve problems involved in a high level activity at the appropriate level of information processing.

The remaining of this paper details the architecture and implementation of the software framework. The next section explains the split of the framework into two big parts (1) sensorimotor control and (2) higher level cognitive processes and introduces mechanisms for data interchange between these two parts. The following sections explain the two parts of the framework in detail. The paper is concluded by a conclusion and an outlook on future work.

II. SYSTEM OVERVIEW

The architecture integrates two different paradigms. First low-level sensorimotor control systems based on adaptive neural controllers are actively developed. The aim is to find robust controllers capable of driving different robotic platforms and to allow higher level cognitive processes to use these behaviors. A low-level neural controller balancing

M. Spranger, C. Thiele, and M. Hild are with the Neurorobotics Research Laboratory, Artificial Intelligence Workgroup, Department of Computer Science, Humboldt-Universität zu Berlin, Unter den Linden 6, 10099 Berlin {spranger|thiele|hild}@informatik.hu-berlin.de

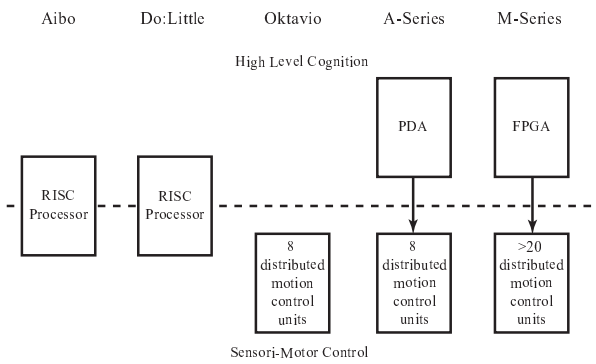
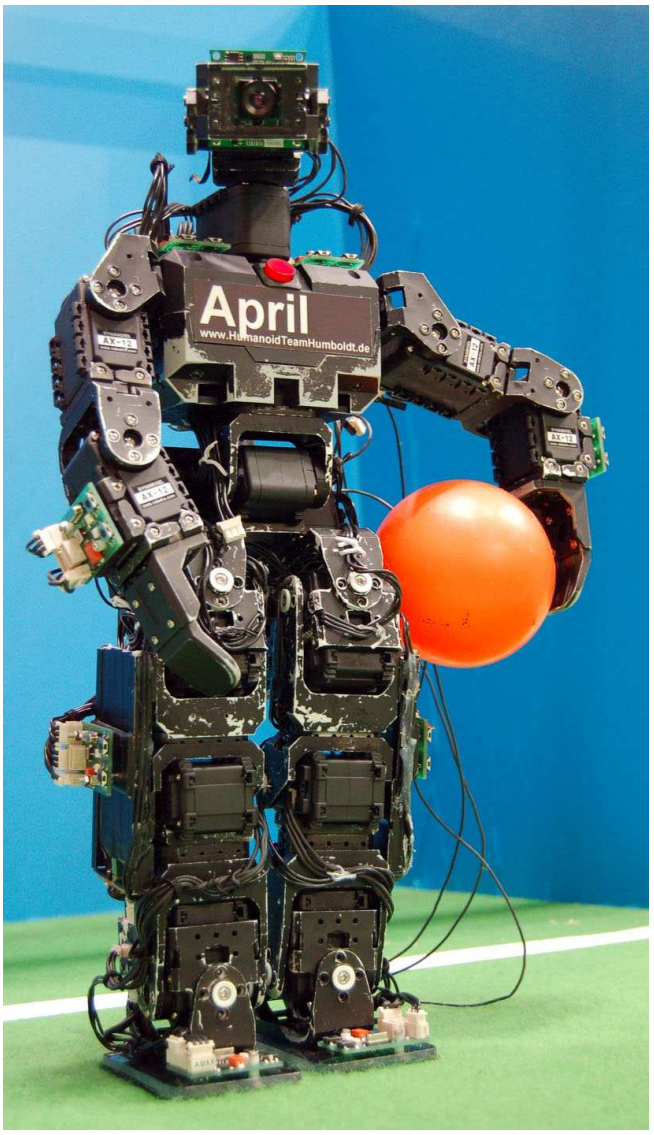


Fig. 1. Systems build using the framework described in this paper. The framework is split into two separate areas of interest (higher level cognition and sensorimotor control), which on different platforms can either be distributed across processes on a single central processing unit or can be distributed over different processing entities. The top row names the different robotic platforms built and under active development.

a two-legged humanoid robot, for instance, may prevent a robot from falling over, while he is trying to grasp an object. The higher level processes control the robots arm and hand, while the balancing controller makes sure that the shift in centre of mass is compensated. Such higher level cognitive processes are organized in a second layer, called the cognitive layer, which is designed for the modularized implementation of such processes as modeling the environment, reasoning and planning and visual perception. Both layers are loosely coupled and allow for the interchange of data in many ways.

The sensorimotor layer has been implemented on a number of platforms, such as the Do:Little, a simple single processor equipped two-wheeled robot, to a full fledged multi-sensor humanoid robot. While the humanoid robot is used for research into complex walking and balancing tasks, the two-wheeled robot is used to investigate the use of evolutionary strategies for creating simple behavior controllers, such as integrating sensory data from distance sensors and wheel encoders into simple obstacle avoidance behaviors. Such basic behaviors also include tropisms, such as light and sound tropisms. In between these two platforms spanning the complexity in terms of degrees of freedom and sensorial configuration we are using a set of platforms to investigate different aspects of sensorimotor intelligence and higher level processes. The eight legged Oktavio robot was developed as a demonstrator for neuronal control of biologically inspired motions. The lab has also used AIBO robots produced by Sony Corporation to investigate higher level cognitive processes in the RoboCup domain. Please see section V for detailed descriptions of all used platforms. Figure 2 introduces all platforms and the distribution of software across processing entities.

This section will explain the two subsystems and their integration given the latest platform built in the lab, called A-



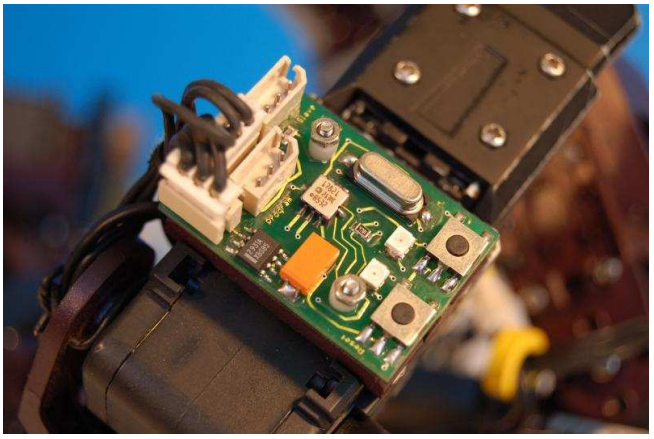


Fig. 2. The main hardware platform running the architecture presented in this paper, the A-series and one of the distributed circuit boards attached to it. The platform is equipped with a PDA in the back executing higher level cognitive processes, the smaller circuit boards are distributed across the body and process sensorimotor data in a tight loop.

series. The A-series was constructed by augmenting a readily available commercial Bioloid robot construction kit [10], [11]. The Bioloid was extended morphologically through specially crafted plastic parts and electronically through special micro processor equipped sensor and actuator control boards distributed across the whole body, a camera and a Siemens Pocket Loox PDA. The original power supply included in the robot kit was replaced by three lithium polymer cells per robot, two at the let and right upper leg limbs and one in the trunk. The eight sensor and actuator control boards, called AccelBoards, all feature simple two axis acceleration sensors, a RS485 motor control bus interface (used to drive the Dynamixel actuators) and a Renesas RISC processor. Together they form a net of distributed processing nodes, that are connected via a shared half-duplex RS485 bus, which is used to share the sensory and motor control data. The AccelBoards are connected to the PDA via a RS232 bus. The PDA connects to the image stream provided by the pan-tilt camera of the robot via a framegrabber device.

The sensorimotor control framework is specifically designed to run on low cost embedded hardware present in even the simplest hardware systems and is tightly coupled with the different hardware architectures. The framework provides a hardware abstraction layer for implementing neural systems on very different processor and hardware architectures. Neural controller can thus be very easily evaluated in different settings and on different platforms. This reflects the general idea of finding robust neural networks specific either via artificial evolution and analysis of its outcomes or via designing neural networks given profound knowledge of underlying systems dynamics, signal processing capabilities and properties of recurrent neural networks. The idea is to use neural networks not only in an evolutionary approach, but also to analyze the evolved controllers and to engineer more complex networks given the obtained results.

The second framework modularizes tasks and allows for independent component development and testing. It provides mechanisms for the distribution of components over processes, transparent inter-component and inter-process communication as well as an elaborate debugging architecture. This framework was and is used to engineer vision driven object and world modeling tasks [12], [13], as well as complex behavioral strategies [14].

Both systems can be loosely coupled across hardware boundaries, which not only facilitates separate development efforts in both frameworks, but via a defined interface enables the group to integrate solutions a posteriori. On A-series robots both systems communicate via a RS232 interface which connects the central processing power of the attached PDA with the distributed processing boards driving basal motor capabilities. The defined interface allows for abstract commands, such as walking with speed, direction and rotation, while retaining possibilities for more fine grained control such as driving concrete controllers with specialized parameters to complete low level motor control exerted by higher level processes. A simple blackboard architecture allows reading and manipulating all data available in the lower level processes. It is this blackboard which is used to communicate between higher level processes and neural motor control. Whether or not and how high level commands are executed is up to lower level motor control processes. The effect and success of higher level actions can be assessed by the higher level processes through direct reading of sensor values. Notice that this in turn enables for a dynamical systems representationless motor control architecture, while still allowing for kinematics to be established by higher level processes, which means that behavior can be overlaid. A grasp movement for instance can be controlled by visual processes and is stabilized by the neural motor control and in case of emergency operations (for instance when the robot is falling) the robot to attain a safety posture. The blackboard also allows the higher level processes to control motors directly.

III. HIGHER LEVEL COGNITIVE PROCESSES / COGNITION

This section deals with the part of the framework used in modeling higher level cognitive processes such as modeling the environment, planning and reasoning. Such processes are predominantly developed for vision equipped robotic agents, that are able to build up larger scale models of the environment, which empower complex planning and reasoning processes.

The software framework used to develop these different processes is a successor of an architecture [15] used in the development of complex, real time, autonomous robotic agents for playing robot soccer in the RoboCup domain. The original system was used quite successfully in that domain and helped to become the World Champion in the four legged league twice and has evolved quite substantially over time [16], [17], [18]. The architecture is specifically aiming to solve the problem of integrating software development groups at different locations. Originally the software was running on top of the OPEN-R middleware (see for example [19] for an early overview), a middleware system transparently connecting components via hard and software boundaries. The middleware was provided by SONY for the AIBO robot platform and used internally by Sony laboratories, but also externally through the RoboCup community. The architecture provides four basic mechanisms

- a mechanism for splitting a task into smaller subtasks (modularization)
- a mechanism for changing the implementation to the problem posed by a subtask during runtime
- mechanisms for distributing modules across processes and a simple inter-communication mechanism
- powerful debugging mechanisms

The main idea of the system is to provide an easy, adjustable apparatus to split a computational task, like playing soccer, into a number of smaller units called *modules*. These units are defined through interfaces, which are sets of data structures. These data structures, called *representations* define the input and output of modules. That is modules are clearly separated abstract components, which allows for

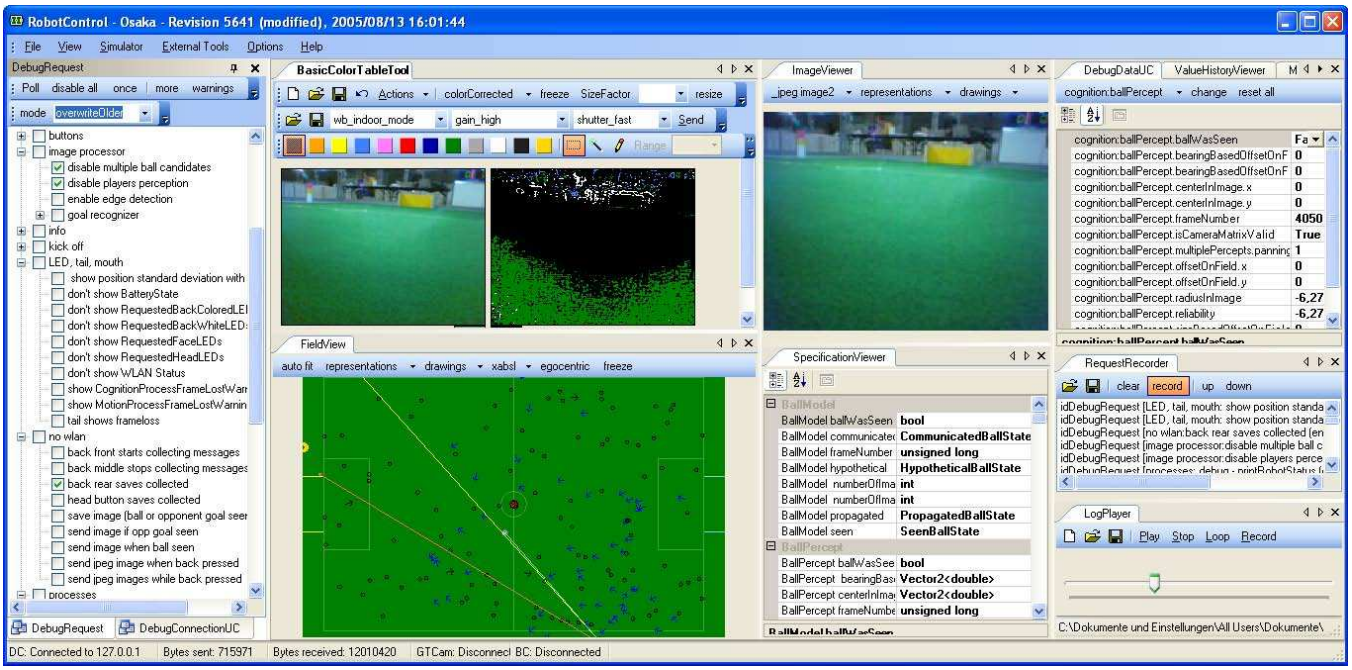


Fig. 4. Tools are an important part of every computational architecture. The image shows the tool used to inspect the internal state of different robots and robot platforms. Depending on the task (robotic soccer, language evolution) developers and researchers have access to a variety of generic debugging mechanisms implemented on all platforms. This includes such things as generic data inspection and modification mechanisms (upper right), generic drawing (middle row, upper images: in-image drawing; lower image: world modeling drawings used in the RoboCup domain) and debug switch mechanisms (panel on the left). The tool is connected to a AIBO robot.

different approaches to a certain subtask to be developed independently and to be compared to other solutions of the same (sub)problem or task. The actual implementation of a *module* is called *solution* and reflects a certain mechanism to solve the problem posed by input, output data structures, which means that given the input data, a *solution* provides an algorithm to compute the output data. Modules therefor allow clear separations of concerns through defined contracts/interfaces. *Solutions* are the different algorithms developed and used separately for solving a certain task (e.g. visual object perception). Together with a mechanism to switch *solutions* during runtime this provides easy testing and comparison facilities for algorithms.

To assist the designer and researcher trying to model a certain system, additional procedures for distributing modules across different processes and communication between processes are provided. A simple message queue algorithm driving an event provider and consumer model is used for inter-process communication. Additionally message queues can be sent via network TCP/IP mechanisms, rendering the connection of processes across hardware devices transparent. Basically processes fill data structures which are part of the interface of one or more modules. Processes are usually triggered by events like an incoming new image from the raw camera, which means that they are tied to hardware level processes. Additionally they may also communicate between each other. The system allows to specify wether the update of a certain data structure causes the process to be run, which is a very flexible mechanism for defining execution times (process relative) and ordering.

The architecture features three main debugging mechanisms (1) debugging switches, which enables the conditional execution of code (2) a generic drawing mechanism and (3) a generic mechanism for manipulating data. These mechanisms are mostly inspired by traditional debugging mechanisms and are an attempt to replace debuggers, which may not be available on all platforms. Debuggers can also cause significant overhead in execution time on some of the platforms especially when used in remote debugging scenarios.

The proposed method of debugging switches, called *debug requests* is used to enable and disable parts of the source code. These switches are a runtime method, which enables developers to switch on and off the execution of certain parts of algorithms. The method is at the core of the debugging mechanisms and is also used to trigger the sending of debugging information used by the two other mechanisms. The switches are implemented in terms of C++ macros consisting of a textual key and the code to be executed conditionally. A runtime system manages the state of the keys and triggers the execution of the associated code parts accordingly. An important feature of the implementation is that the runtime system only manages debug states that are visible and allows connecting debugging tools to query the available keys. This enabled us to develop debugging tools which are to a certain degree generic, that is it does not matter which robot platform it connects too, but only the support of these mechanisms is relevant.

Next to debugging switches the architecture provides a way of drawing geometric figures in the robot control code. The main idea here is that visualizations are a very useful

Fig. 3. Other hardware systems empowered by the architecture. From top to bottom: the AIBO robot used in RoboCup, the eight legged Oktavio walking machine and the Do:Little platform..

debugging tool and are typically algorithm specific, which makes it useful to bundle actual code and interleave it at the relevant places with visualizations. Since most of the platforms do not come with a display, and those that do (A-series) in principle can use the same system to separate the visualization from the actual robot control code, the visualization is usually run on an accompanying laptop or PC used for debugging. The mechanism is implemented using macros, which trigger the sending of debug messages. The debug messages include an identifier of the shape to be drawn and a variable number of parameters depending on the shape (for instance four parameters: x, y, width and height for a rectangle in the image) and the space the shape is drawn too. Two coordinate systems are supported, one for drawing into images and one for drawing objects relative to the robots body on the ground. Next to this vector graphics approach, a mechanism is available which allows for direct pixel manipulation of images. The sending and manipulation of these images is implemented similar to the drawing mechanisms via C++ macros, which consist of a textual id and some primitive drawing operations.

To be able to manipulate data in a generic way a data manipulation scheme has been developed. Data can be exchanged through the message passing and queuing for inter-process communication. On top of the serialization of objects into message and message queues we developed a mechanism that gathers a additional information about variable names and class inheritance hierarchies. The main idea is that during serialization of objects, a specification of the serialization is built up, which is later used to create generic data manipulation and drawing dialogs. The specification in itself is a description of serializable objects on a byte level. That is every object serialization can be decomposed into primitive data types (such as *int* or *double*) with names. The mechanism is implemented as C++ macros, which are used inside the *serialize* functions, which govern the serialization process. For this to work in complicated inheritance hierarchies, all objects in the hierarchy have to use these macros.

All debugging techniques are easily accessible through an external debugging tool which connects to the robot processes and allows for the conditional execution of code on the robot (via debug switches), inspection of debug drawings and provides dialogs for manipulating and plotting data (see figure 4). It also offers facilities for recording log files, replaying them and even sending them to the robot hardware acting. The latter of which is helpful in oracle scenarios, where the robot is fed with artificial images or artificial world models and so forth.

IV. SENSORIMOTOR CONTROL SYSTEM / SPINALCORD / HARDWARE CONTROL SYSTEM

In earlier robotic systems built and operated in the lab, the sensorimotor control system was separated from the higher level cognitive processes through thread or process borders. On newer hardware like the A-series, these processes are split onto independent hardware. The aim is to have a

sensorimotor control, which is cheap but achieves guaranteed execution times. In our view this can best be accomplished using a distributed architecture, with small special devices each handling a subpart of the overall task without adding the overhead of process and task scheduling. The small and cheap circuit boards distributed across the body of the A-series for example, each control a few motors and read the values of a few sensors. But they do so in a very reliable and fast manner.

The distributed processing entities provide the data (motor control and sensory data) on a shared communication bus, called *SpinalCord*. The bus is updated with a frequency of 100 Hz, which is also the frequency for calculating new motor control data. However, sensor data like the data stemming from the two dimensional acceleration sensors is acquired more often and median filtered before data exchange.

The communication baud rate differs, but is at least one MBaud with at least 256 bytes of data shared per time frame. On newer systems like the currently designed M-Series humanoid robot we will increase communication speed. The communication protocol is exactly timed, so that defective boards are handled gracefully. No data is transmitted from the defective board and all other boards are using the last received data. If e.g. one arm on a humanoid robot malfunctioning, the rest of the system will stay operational. This makes the system very reliable. An example of such an architecture is the *Oktavio*. Legs can be added and removed from the body trunk, while the robot is operating. The robot can be repaired at runtime by changing legs, but the same principle is used to investigate different configurations of legs.

Luckily having a lot of distributed circuit boards does not mean one has to develop more software. In a sense the boards constitute a distributed but homogenous platform, since every board runs the same firmware. However, in some cases it makes sense to make the boards location aware. The position of a board on the body is detected using the attached hardware, e.g. on the A-Series humanoid robots each actuator has an unique ID, which can be queried and serves as an indicator of the board's position on the body.

The SpinalCord data is read- and writable from high-level cognition processes. So one way of controlling an actuator is by using the high-level cognition framework introduced in the previous section. This is used on the A-Series humanoids for controlling the pan and tilt unit of the head (which contains the camera).

The simplest way for on-board motion control is using a keyframe technique. We are using one keyframe motion net at a time, which encodes all needed motions. It contains branching points for changing the currently played motion using a *selector* (which is only one simple ASCII byte). Building motions using the keyframe technique is simple and fast.

Without using sensor data, unstable systems i.e. humanoid robots are difficult to control. The architecture therefore offers the possibility to control motions using a bytecode language. Depending on the processor used, this language has to be more or less simple. We at the Neurorobotics Research Laboratory are using a neural bytecode describing recurrent neural networks. With this language it is possible to control the whole system using neural networks.

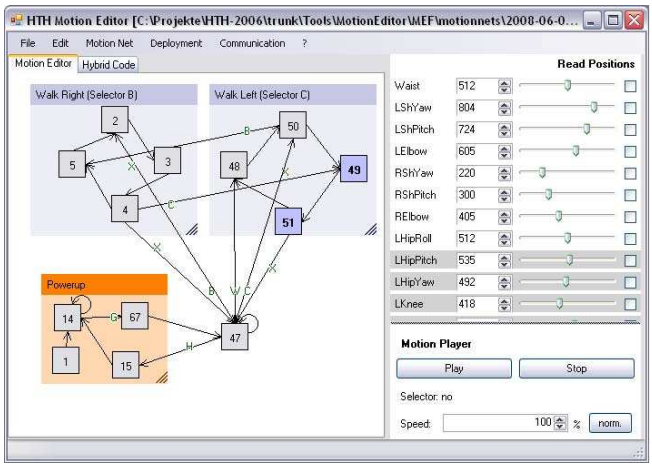


Fig. 5. The MotionEditor software with a simple motion net on the left and corresponding motor values on the right

The two techniques could be merged in different ways. It is possible to have some motions in keyframe technique and some using neural networks. Furthermore it's also possible to let the keyframe technique calculate the next motor control values and then adjust them using neural networks. These are able to include sensor data and so it's even possible to stabilize keyframe motions.

Having these two techniques one has broad possibilities for motion control: from simple and fast to develop keyframe motions to robust and stabilized neural motions. So keyframe motions could be used to easily analyze motions and then develop motions using neural networks. It's also possible to convert keyframe motions to neural networks using the neural one-shot introduced in [20].

Creating keyframe motions and linking them to neural bytecode is made easy using a custom software called *MotionEditor* (see figure 5). Keyframe nets are created using a simple drag-and-drop interface. Using the syntax-highlighting editor one is able to write neural "programs" and to link them to the keyframe net. Keyframe values could be manually adjusted or the robot is put into a desired posture and the values are read out. Therefor the robot is directly linked to the PC using a standard USB connection. With only one click it is possible to deploy keyframe nets into the processor's memory. The robot can then act autonomously.

The introduced simple bytecode is used on different hardware platforms and is even implemented for PCs using Windows and Linux for simulation purposes. In the project *e-Robot* it will be possible to control simple robotic experiments using such bytecode through the internet and watching the results as a video.

This architecture is cheap and robust and offers keyframes and neural networks to control the actuators. Like in real humans and animals, the motion control is separated from the

high-level cognition and e.g. reflexes could be implemented without changing the high-level part. The cognition only asks for things like *walk left* and the motion control is highly independent on how it achieves this. The high-level cognition could also set values of input neurons and then ask for *walk* and the robot walks into the direction corresponding to the input value.

V. Additional Hardware Platforms

The architecture presented here has been used on different platforms, with the already introduced A-series robots being the most complex. However, we have also used other platforms aimed at very specific research question, among them small Do:Little robots.

The Do:Little is a two-wheeled robot designed for mass production. It is equipped with a 20 MHz CISC processor featuring additional DSP technology. The platform can sense its environment using three brightness sensors, five infrared distance sensors and two active ground gradient sensors, which make the platform suitable for evolving neural controllers in obstacle avoidance tasks and light tropisms. Additionally, the robot features two microphones to investigate phonotaxis and swarm behaviors. To study social behaviors in groups the perimeter of the robot is bronze covered and silver contacts are spread throughout the body which allow robots to exchange energy. Special electronic circuit design lets robots decide wether they want to act as an energy source or sink.

One of the first bigger robots built to investigate walking controllers was the Oktavio. Oktavio is designed to be a universal platform for evolving walking machines with different configurations of legs. Up to eight legs can be added and removed while the robot is operating. Each leg is an autonomous energy and processing entity, equipped with a 20 Mhz CISC processor, three joints powered by a set of motors and electronic component as well as energy supplying batteries. The legs can sense their environment and state through motor encoders measuring the state of motors, two infrared distance sensors, as well as a specially constructed foot-ground contact sensor, which measures the exact ground contact position and force of the leg. The body construct, a plastic plate has eight connectors which allows for up to eight legs to be added and a ring bus system allowing legs to exchange signals and align their walking control patterns.

Another research avenue long pursued in the lab is the RoboCup domain. One of the leagues in RoboCup uses AIBO robots produced by Sony to investigate the integration of perception, modeling, reasoning and planning and action architectures and algorithms grounded in a dynamic environment of soccer playing robots. The robots are equipped with a movable camera, distance sensors, as well as microphones and a speaker. They can interact with the environment using four legs, each with three degrees freedom and a foot-ground contact sensor. All motors driving the degrees of freedom are equipped with angular encoders. The platform is the starting point for the part of the architecture integrating vision and proprioceptive sensor data to drive complex environment modeling processes which are used by behavior processes to drive actions such as walking and shooting.

In an ongoing effort to combine experiences made with all previous platforms, a new humanoid robot is developed from scratch. This M-series is a successor of the A-series and tries to scale the solutions developed for earlier platforms to a much bigger robot. The robot will be about 1.20m tall and is integrating experiments made in phonotaxis, speech synthesis and recognition, motion balancing and two-legged walking controllers with visual processing of a pan tilt camera in the head. In addition it will feature two actuators at the end of each arm for picking up and placing of objects, allowing for complex interactions with the environment.

VI. Conclusions and Future Works

This paper presented a loosely coupled software framework used in very different research scenarios. We have shown how the framework is used on multiple platforms, allowing researchers to quickly adapt and integrate solutions and algorithms developed. Additionally we showed how to modularize specific parts of the architecture and how to enable researchers and developers to interact with hardware centric systems. Future work will mainly concentrate on integrating the past research experience on a new hardware platform, the M-series.

VII. Acknowledgements

This research has been carried out at the Neurorobotics Research Laboratory, which is part of the Artificial Intelligence Workgroup at the Humboldt-Universität zu Berlin with partial support from the ALEAR project, funded by the EU Cognitive Systems program. The information provided here is the sole responsibility of the authors and does not reflect the EU Commission's opinion. The Commission is not responsible for any use that may be made of data appearing in this publication.

The authors would like to thank all members of the Humanoid Team Humboldt who have worked on previous versions of the software and who are filling the architecture with life. We are also indebted to architects of the GermanTeam contributing to the cognitive layer especially Matthias Jüngel and Martin Löetzsch.

References

[1] Steels, L.: Evolving Grounded Communication for Robots. Trends in Cognitive Sciences **7**(7) (2003) 308–312
[2] Steels, L.: Semiotic Dynamics for Embodied Agents. IEEE Intelligent Systems Summer Issue (2006) 32–38
[3] Steels, L., Spranger, M.: The robot in the mirror. Connection Science **20**(4) (2008)
[4] Steels, L., Lötzsch, M.: Perspective alignment in spatial language. In Coventry, K.R., Tenbrink, T., Bateman, J.A., eds.: Spatial Language and Dialogue. Oxford University Press (2008) To appear.
[5] Wellens, P., Lötzsch, M., Steels, L.: Flexible Word Meaning in Embodied Agents. Connection Science **20**(2) (2008) 173–191
[6] Thórisson, K.: Integrated AI Systems. Minds and Machines **17**(1) (2007) 11–25
[7] Pfeifer, R., Scheier, C.: Understanding Intelligence. MIT Press (1999)
[8] Pfeifer, R., Lungarella, M., Iida, F.: Self-organization, embodiment, and biologically inspired robotics. Science **318** (November 2007) 1088–1093

[9] Kitano, H., Asada, M., Kuniyoshi, Y., Noda, I., Osawa, E.: RoboCup: The Robot World Cup Initiative. Proceedings of the first international conference on Autonomous agents (1997) 340–347
[10] Hild, M., Jüngel, M., Spranger, M.: Humanoid team humboldt team description 2006. In Lakemeyer, G., Sklar, E., Sorrenti, D., Takahashi, T., eds.: RoboCup 2006: Robot Soccer World Cup X Preproceedings, Bremen, Germany, RoboCup Federation (2006)
[11] Hild, M., Meissner, R., Spranger, M.: Humanoid team humboldt team description 2007. In Visser, U., Ribeiro, F., Ohashi, T., Dellaert, F., eds.: RoboCup 2007: Robot Soccer World Cup XI Preproceedings, Atlanta, USA, RoboCup Federation (2007)
[12] Jüngel, M., Hoffmann, J., Lötzsch, M.: A real-time auto-adjusting vision system for robotic soccer. In Polani, D., Browning, B., Bonarini, A., eds.: RoboCup 2003: Robot Soccer World Cup VII. Volume 3020 of Lecture Notes in Computer Science., Springer (2004) 214–225
[13] Spranger, M.: World models for grounded language games. Diploma thesis. Humboldt-Universität zu Berlin (2008)
[14] Lötzsch, M., Bach, J., Burkhard, H.D., Jüngel, M.: Designing agent behavior with the extensible agent behavior specification language XABSL. In Polani, D., Browning, B., Bonarini, A., Yoshida, K., eds.: RoboCup 2003: Robot Soccer World Cup VII. Volume 3020 of LNAI., Springer (2004) 114–124
[15] Röfer, T.: An Architecture for a National RoboCup Team. **2752** (2003) 417–425
[16] Röfer, T., Brunn, R., Dahm, I., Hebbel, M., Hoffmann, J., Jüngel, M., Laue, T., Lötzsch, M., Nistico, W., Spranger, M.: GermanTeam 2004. In Nardi, D., Riedmiller, M., Sammut, C., Santos-Victor, J., eds.: RoboCup 2004: Robot Soccer World Cup VIII Preproceedings, Lisbon, Portugal, RoboCup Federation (2004)
[17] Röfer, T., Brunn, R., Czarnetzki, S., Dassler, M., Hebbel, M., Jüngel, M., Kerkhof, T., Nistico, W., Oberlies, T., Rohde, C., Spranger, M., Zarges, C.: GermanTeam 2005. In Bredenfeld, A., Jacoff, A., Noda, I., Takahashi, Y., eds.: RoboCup 2005: Robot Soccer World Cup IX Preproceedings, Osaka, Japan, RoboCup Federation (2005)
[18] Röfer, T., J., B., Carls, E., Carstens, J., Göhring, D., Jüngel, M., L., T., Oberlies, T., Oesau, S., Risler, M., Spranger, M., Werner, C., Zimmer, J.: GermanTeam 2006. In Lakemeyer, G., Sklar, E., Sorrenti, D., Takahashi, T., eds.: RoboCup 2006: Robot Soccer World Cup X Preproceedings, Bremen, Germany, RoboCup Federation (2006)
[19] Fujita, M., Kageyama, K.: An open architecture for robot entertainment. Proceedings of the first international conference on Autonomous agents (1997) 435–442
[20] Hild, M., Kubisch, M., Göhring, D.: How to Get from Interpolated Keyframes to Neural Attractor Landscapes – and Why. 3rd European Conference on Mobile Robots (2007)

Ikaros: Building Cognitive Models for Robots

Christian Balkenius, Jan Morén, Birger Johansson and Magnus Johnsson

***Abstract*— The Ikaros project started in 2001 with the aim of developing an open infrastructure for system-level brain modeling. The system has developed into a general tool for cognitive modeling as well as robot control. Here we describe the main parts of the Ikaros system and how it has been used to implement various cognitive systems and to control a number of different robots ranging from robot arms and hands to active vision systems and mobile robots.**

I. INTRODUCTION

The goal of the Ikaros project is to develop an open infrastructure for system level modelling of the brain including databases of experimental data, computational models and functional brain data. The infrastructure supports a seamless transition from a pure modelling and simulation set-up to real-time control systems for robots running on one or several computers in a single or multiple threads. Computational models are built by connecting individual modules that implement a specific brain model or algorithm into larger systems.

The system makes heavy use of the emerging standards for Internet based information such as XML and makes all part of the system accessible through an open web-based interface. We believe that this project has the potential to radically change the way system level modeling of the brain is performed in the future by defining standard benchmarks for brain models and substantially increase the gain from cooperative research between groups.

A system like Ikaros can not operate in a vaccuum. Instead, the goal is to allow Ikaros to easily work with as many external sources of information as possible. There is simply too many types of information that need to be used by the system and without taking an inclusive approach, the task of adapting information and models becomes too great. The only viable solution is to integrate Ikaros with other similar endeavors whenever possible. This inclusive approach means that we want to offer a large corpus of experimental data from cognitive experiments for use with Ikaros, but we also strive to make it easy to adapt other experimental data for use within the system.

Inclusivness also means making development a transparent and straightforward process. As part of the standard infrastructure, Ikaros already contains a sizable number of standard modules that are useful in a broad range of cognitive models. The infrastructure also contain modules that allow for an easy interface with various types of hardware such as video cameras and robots. For example, there are easy interfaces to the various standards for video capture and video files, for audio processing as well as for robot control through a set of drivers for different hardware systems.

The goal of the infrastructure specification is to be minimally demanding for anyone developing an Ikaros module. It should be possible to learn to use it in a few minutes while still providing support for very complex architectures. In the following sections we describe the different parts of the Ikaros system and the choices that have been made when designing the different components.

C. Balkenius, B. Johansson and M. Johnsson are with Lund Univeristy Cognitive Science, Kungshuset, Lundagård, SE-222 22 Lund, Sweden. christian.balkenius@lucs.lu.se

Jan Morén is with Knowledge Creating Communication Research Center, NICT, 2-2-2 Hikaridai, Seika-cho, Soraku-gun, Kyoto 619-0288, Japan jan.moren@gmail.com

II. SYSTEM-LEVEL MODELS

The core concept of system-level modeling is the module which corresponds to a part of a model. A module can have a number of inputs and outputs and encapsulates a particular algorithm (Fig. 1). This does not mean that cognitive models built using Ikaros must adhere to a modular view of cognition. Instead, a system-level approach to cognitive modeling acknowledges that different cognitive components interact in many ways and it is one of the strengths of the approach that it explicitly shows these interactions as connections between modules. A module in Ikaros is thus not a statement about locality or impenetrability, it is only an acknowledgement that a system is constructed from several components, and these components or modules have different properties.

In general, to design a system-level model it is necessary to answer four questions:

What are the components of the system? This entails answering at what level the model should be described. Are the components individual neurons or brain regions, or are they some form of abstract description of functional components without direct relation to the brain? There is no single correct answer to these questions; it depends on the model being implemented.

What are the relations between the components? Are they parallel systems with little interaction, or are they tightly coupled? Are they all at the same descriptive level or are some components subparts of others? Is the system heterogeneous or hierarchical?

Which function is performed by each component? How can the functions be described as mathematical functions or as algorithms? Ikaros supports systems built from standard modules that implement elementary mathematical functions as well as modules that are hand coded from scratch.

What information is transmitted between the components and how is it coded? The question of coding is the most important for a system-level model and the only one where

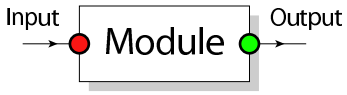


FIG. 1: *A module with one input and one output.*

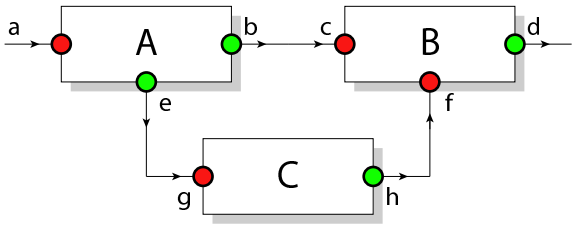


FIG. 2: *A small system with three modules A, B, C with connections between them.*

Ikaros puts any major constraints on the possible models. In Ikaros, all inputs and outputs are coded as matrices of floats. This limits the possible models in several ways that make it more likely that different models can be interconnected. Although Ikaros puts no constraints on the interpretation of the matrices, this type of structure is best used for coding in terms of numerical values, either directly or using some form of distributed code.

In Ikaros, the components are specified using an XML-based language which also describes the relation between the components. The function in each component is described either using standard modules or by writing new simulation code. The transfer of information between components is implicit in the coding of the different modules.

III. DESCRIBING MODELS

Fig. 1 shows a simple module. This module has a single input through which it receives input data and a single output through which it sends its output data. The input is read in discrete time and the module also generates new output at discrete intervals.

Modules can be connected together to form systems (Fig. 2). This network of modules is what makes up a model in Ikaros. Here, the model consists of three modules A, B and C. Module A has one input (a) and two outputs (b and e). Module B has two inputs (c and f) and a single output (d). Finally, module C has one input (g) and one output (h). The complete model has the single input a and the single output d.

One of the greatest strengths of Ikaros is its ability to handle large complicated cognitive models consisting of many interacting subcomponents. To allow the specification of such architectures, an XML-based description language has been developed [6]. This language has three main components: the module, the group and the connection.

A module element describes an instance of a particular Ikaros module and sets its parameters. These parameters are handled to the constructor function of the module as described below. The only two required attributes are *class* and *name* that decides what code the module will run and how it will be referred.

```
<module
    class = "MyClass"
    name = "MyModule"
    alpha = "3"
    beta = "0.1"
/>
```

A connection between two modules is specified in a connection element:

```
<connection
    sourcemodule = "Thalamus"
    source = "Output"
    targetmodule = "Amygdala"
    target = "Input"
/>
```

Finally, it is possible to group modules and connection in to larger structures. The following example corresponds to the structure shown in Fig. 3 and Fig. 4. It defines a group (or new module) called X with an input x and an output y. The group consists of three modules A, B and C which have multiple connections between them. The input x is connected to the input a of module A and the output y receives data from output d of module B.

Groups can also be given inputs and outputs to let them function as new modules or be read from external files and be used as call descriptions. A specification of these features is however beyond the current description.

IV. THE SIMULATION SYSTEM

Currently, the main part of Ikaros is the simulation system which consists of a platform independent simulation kernel together with a large set of modules that implements different functions and models.

A. Design Criteria

There were a number of important considerations in the choice of the simulation structure. The first was that it should be platform independent. There are two reasons for this. The first is that it was expected that the system would be required to run on different architectures. The second, and more important reason was that the we did not want to depend on one particular compiler or operating system. It is well known that code is only portable once it has been ported. By simultaneously developing for several operating systems, it would be almost guaranteed that Ikaros would be reasonably portable. We have consequently strived to comply with the relevant standards as much as possible. These includes ANSI C++, POSIX and BSD sockets. A related choice was to depend on as few external libraries as possible. Although the current version of Ikaros uses external libraries for sockets, timing, threads and mathematical operations, it can still be run in a minimal version that only uses a small set of standard C++ libraries.

The second main design choice was to use a discrete-time model for simulation. Although this is the normal operation for most neural network simulators, there are some notable exception. However, to allow the easy integration of different

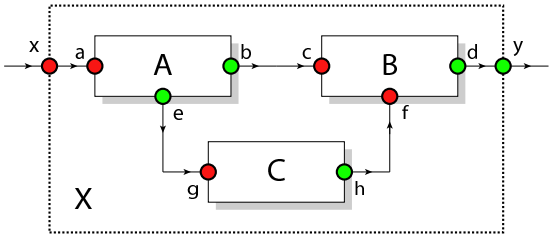


FIG. 3: *A group consisting of three modules. The group is externally considered as a module named X with one input x and one output y. These inputs and outputs are internally connected to input a of module A and output d of module B.*

```
<group name = "X">
   <input name = "x" targetmodule = "A" target = "a" />
   <output name = "y" sourcemodule = "B" target = "d" />
   <module name="A" ... />
   <module name="B"  ... />
   <module name="C"  ... />
   <connection sourcemodule= "A" source = "b" targetmodule = "B" target= "c" />
   <connection sourcemodule= "A" source = "e" targetmodule = "C" target= "g"/>
   <connection sourcemodule= "C" source = "h" targetmodule = "B" target= "f" />
 </group>
```

FIG. 4: *Example of a group of modules with its own input and output. The graphical representation of this system is shown in Fig. 3*

types of algorithms, it was decided that a discrete time simulator would be most useful. It is hard to imagine how many algorithms could be adapted to a continuous time framework. In most cases, this choice does not limit the possible models that can be designed since it only relates to the times when different modules communicate and not their internal structure.

Another consideration was that to make the system attractive it should be as easy as possible to use many different types of programming styles. As a consequence, we decided to only use standard C data structures such as integers and matrices of floats. The use of doubles was decided against on grounds of efficiency and the lack of support for doubles in most vector co-processors.

B. Module Interface

All inputs and output of modules are represented as arrays or matrices of floats and the sizes of these matrices are represented by integers. The sizes of all data structures used by Ikaros are calculated during startup and can not be changed during execution. This restriction only applies for the data moved between modules; for internal data used in modules there are no restrictions at all. The actual code in a module can use any coding style as long as the inputs and outputs are in the right format - indeed, it is entirely feasible to embed or interface with an interpreter in a module for a completely different language transparent to Ikaros itself. Since Ikaros itself is written in C++, either C like or C++ like coding styles can be used as long at it is wrapped in a C++ class. Although the inputs and outputs are part of the Ikaros kernel data structures, the modules themselves does not know about this. Instead, they can magically assume that the input matrices are always filled with the required data. This design decision has made it easy to incorporate code not specifically written for Ikaros as long as it is reasonably clean. For example, the main function of a trivial module that would only copy its input to its output may look like this:

```
MyModule::Tick()
{
   for(int i=0; i<size; i++)
      output[i] = input[i];
}
```

The point here is that this code looks like any C++ code and there is nothing Ikaros specific with it. When this function is called, the array input will contain the input to the module and after execution, Ikaros takes care of the result in the array output.

It was also considered fundamental that simulations using Ikaros would not be slower than simulations made in a dedicated system. Conceptually, all modules in Ikaros run concurrently and synchronously. This mode of operation was selected because it is the only possibility when it is necessary that execution order is well defined, which is the case for many algorithms. Because of the synchronous operation, there will be a delay of exactly one time step (or tick) between the production of an output from a module and the time when it can be used by another module. In most cases, this extra copying step is necessary anyway and does not usually incur any extra execution cost.

Since this overhead is not always desired however, version 0.8.0 introduced zero-delay connection between modules. Using this type of connections, there is no delay at all between the production of an output and its use by other modules. Instead, the second module refers directly to the

memory where the first module has produced its output. To make the result well defined, zero-delay connections are only allowed within subsets of the complete module networks that form directed acyclical graphs. That this condition is fulfilled is checked during start-up when all modules are sorted according to their position in the graph. With zero-delay connections, the input to the system can in principle be processed in a single time step regardless of the number of modules that the information passes on its way to the output. In this case, the execution overhead is negligible.

The kernel also includes a small set of libraries that hides system specific code for sockets, timing, threads and serial communication. In addition there are utility libraries for memory management, XML processing and mathematical functions. In most cases, the programmers need not know about any of these libraries to use Ikaros.

C. Kernel Start-Up

The kernel is responsible for the creation of the network and its modules at startup, the scheduling during system execution, and the propagation of data between modules. Fig. 5 shows the main component of the running Ikaros system.

Detailed knowledge of the kernel operation is not at all necessary or even recommended for use of Ikaros. Knowing why and in what order things are started do however make it easier to understand the design decisions made. This section can be skimmed lightly without any loss of understanding.

The most important aspect of the kernel is the creation sequence that occurs when the system starts up. This happens in six steps:

a) Class Registration: When the Ikaros program starts, it first registers all code for the modules contained in the system. This initialization step builds a data structure that contains pointers to a creator function for each module type and binds it to a module class name.

b) Module Creation: When the initialization has finished, the kernel reads the supplied control file in XML-format, which specifies the modules to activate and gives them instance names and other parameters. One instance of each module specified is created for every occurrence of that module in the control file. A module can thus have multiple instantiations with different parameters. When each module is created, it registers its inputs and outputs in the kernel to allow them to be connected in the next step. At this stage, the individual modules also gain access to any additional parameters set in the control file for that particular module.

c) Connections: When all modules have been created, the kernel continues to read the control file and make the specified connections between modules.

d) Size Calculations: Most input and outputs have dynamical sizes that are set during start-up. For example, if the input of a module is connected to the output of another module that produces a 4x4 matrix, the input of the second module will adapt to this and set the size of its outputs accordingly. There can be any relation between the size of an input and the size of an output.

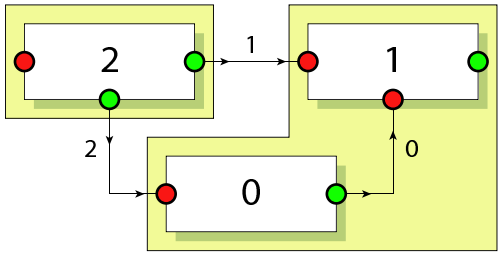


FIG. 6: *The order of execution of three modules. The numbers on the connections indicate the delay in the connections. The numbers on the modules indicate the order in which they should be executed. The two shaded areas correspond to two thread groups.*

For example, the output from the module could be set to have the double size of the input or some other more complex relation. Since there can be a number of cyclical relations between different modules, the calculation of output sizes is performed iteratively until all sizes have been established. If there are cyclical dependencies, these will be found during this stage and an error message will be produced.

e) Sorting the Modules: All modules are sorted in two ways (Fig. 6). The modules are partitioned into different sets that each contains a directed acyclical graphs (DAG) of modules with zero-delay connections between them and only delayed connections to any other modules. Each of these sets can be run in a separate thread and is called a thread group. A topological sort is performed on the groups according to their positions in the DAG which defines a partial order relation on the modules. For modules that have zero-delay connections between them, this order is used to make sure that a module that produces data that another module will use is always executed before that other module.

f) Module Initialization: When all modules have been connected, the initialization phase starts. At this stage, the size of the input that each module will receive is known and each module is allowed to create any additional storage that it needs and initialize variables. To do this, the kernel calls an initialization function for each of the created modules.

D. Kernel Operation

The scheduling mechanism of the Ikaros kernel is responsible for calling the code of each module instance once during each discrete time step (or tick).

In the simplest case, the scheduling consists of calling the tick function for each module in the order in which they were sorted during initialization. When Ikaros runs in threaded mode, each thread group is handled separately in this way. In threaded mode, there is no communication between modules in different DAGs during this time which greatly simplifies the operation of the kernel.

In a second step, the data propagation function is called to copy data from outputs to the inputs of the modules. Data propagation is done simultaneously for all modules. The output for each module is copied to the input to which it is connected. The propagation process is also responsible for the simple data translation that is made by the system and

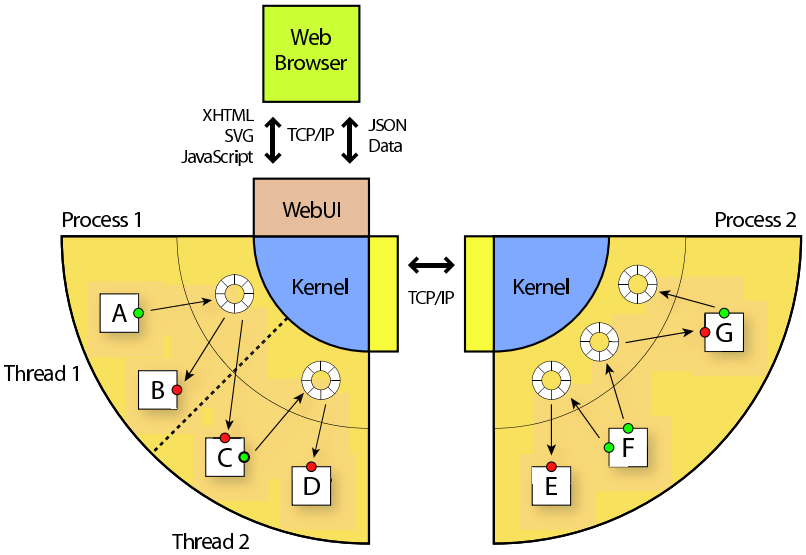


FIG. 5: *The Ikaros kernel. The kernel starts a number of threads where a number of modules (A-G) are executed. The modules communicates through a set of circular buffers that correspond to outputs from the modules. The kernel can also communicate with other Ikaros processes running on the same or on a different processor or computer. In addition, the kernel communicates with an optional graphical user interface client running in a web browser.*

concatenation in the case when several outputs are connected to the same input. In addition, this stage delays the data on connections when this is set in the connection.

Finally, the kernel handles timing when Ikaros runs in real-time mode. In this case, the kernel makes sure that the execution of the tick did not take longer than allowed and waits for the appropriate moment to start the next tick.

E. Anatomy of a Module

Every module in Ikaros implements five functions. For a module named MyModule, the following functions are be defined and called in the following order:

MyModule() The creator function registers all the inputs and outputs of a module. It also gains access to all parameters of this instance of the module from the control file.

SetSizes() This optional function is called repeatedly during start-up to calculate the sizes of dynamic outputs based on the sizes if the inputs to the module.

Init() The init function is called after kernel initialization and lets the module gain access to its inputs and outputs. This is also were any internal data structures are allocated.

Tick() The tick function is where the actual work is being done by the module. It is called repeatedly during the execution of a module and should calculate new outputs based on its inputs (See example in section 3.1).

~MyModule() This optional function deletes any module specific memory that has been allocated in Init() and performs other clean-up that may be necessary.

A template for new modules is available as part of Ikaros. This template is named MyModule and a new module can easily be added to Ikaros by simply renaming the template.

V. STANDARD MODULES

Ikaros contains a large number of standard modules. These can be divided into a number of categories.

IO Modules: There is a set of modules that read data from different file formats, for example text data or different media files. Other modules are used to communicate with external devices such as cameras or robots.

Utility Modules: To simplify the design of models, there are also a large number of utility modules for simple mathematical operations. This includes vector and matrix operations and standard mathematical functions. Other utility modules are used to collect data or statistics or to control an experiment. A few utility modules are used to generate input such as the function generator.

Image Processing Modules: Another set of modules implement standard image processing functions. There are modules to transform the colors in an image, modules that scale images in different ways or performs other spatial transforms. To apply different image processing operators there is a module for convolution, but also modules for specific operators such as the Sobel operator and parametrically defined Gabor filters. There are also several modules that performs edge detection. A few vision modules are more complex and implements a saliency map or an attention focusing mechanism.

Environment Modules: To allow simulation of an agent in an environment, there are a number of modules that implements simple environments. The GridWorld module implements a two-dimensional environment consisting of a grid with obstacles together with an agent that can navigate in it while being controlled by other Ikaros modules. There is also a variant where the simulated robot can move continuously over the grid. This module also simulates a 2D visual field using a ray casting algorithm. Another module simulates an arm with arbitrary geometry.

Other Modules: The standard modules also include a few neural network algorithms and some general learning

algorithms.

VI. REAL-TIME EXECUTION

When Ikaros is used to control robots it is necessary that the precise timing of input and output can be controlled. To accomplish this the kernel has functions to time the execution of each tick. When Ikaros starts up it sets it time-base to the required interval and tries to time the ticks to this time-base. It internally controls that it is able to keep up with the desired speed and will report delays in the execution.

Obviously, the accuracy of the timing will depend on the underlying operating system. The real-time functionality is based on POSIX.4 [21], but since Ikaros is currently not running on real-time operating systems, any other process can in principle interfere with real-time execution. In practice, it is possible to get less than 1 ms resolution on the operating systems we have tested.

An important factor that contributes to real-time performance is the ability to run Ikaros in multi-threaded mode [10]. In this mode, the kernel tries to run every module in a separate thread. When there are zero-delay connections between a set of modules, the kernel will automatically put these in the same thread.

In thread mode, each module can be set to run at different time intervals. For example, a slow visual processing module may run 5 times per second while a faster motor control module can be allowed to run 100 times per second. This feature is very useful for robotic control where some loops need to run at high speed while others are much heavier.

VII. A GRAPHICAL USER INTERFACE

To monitor ongoing simulations, Ikaros has a graphical user interface. Like the modules and connections, this user interface is specified using XML. This XML specification is read by the Ikaros kernel which starts up an integrated web-server which allows standard web browsers to act as graphical clients. The browser gets a set of JavaScript routines from Ikaros that are run in the browser that implements the graphical user interface [9]. The actual drawing is made using SVG [8]. The choice of JavaScript+SVG was based on the fact that this would make the system truly platform independent.

For communication with the sever, the interface uses JavaScript Object Notation (JSON). Although we initially planned to use XML for this communication, JSON turned out to be much simpler to use since it can be natively parsed by JavaScript using the eval function.

Unfortunately, few browsers initially supported SVG and we made the choice to only actively support FireFox. The first version of Ikaros that used this graphical user interface was released a few days before the first version of FireFox to include native SVG rendering (version 1.5). Today, several other browsers support SVG and JavaScript in the required way including Safari and Opera.

Currently, Ikaros has support for graphical objects such as bar graphs, different forms of 2D and 3D plots, images, grids and vector fields. The graphical client can easily be extended with new graphical objects by writing JavaScript code for the drawing of the new object.

One limitation of this solution is that it is not as fast as using a dedicated program for the client. However, we felt that this solution has several advantages. First of all, it means the whole system becomes totally platform independent. But also, and perhaps more importantly, it enables us to transparently monitor and control a running simulation remotely, independent of what system the simulator and the client is running, and we can do so with a simulation running in another room or across two continents with no loss of functionality.

If fast, concurrent representation is important, the very open-ended structure of an Ikaros module enables users to simply write a graphical module that includes the toolkit or other representational system of their choice and display data sent to the module from there. Likewise, a module that receives user interaction can change the behavior of other modules in the system accordingly by defining a ”command channel” that sends data to other modules via the same mechanism as ordinary data. Ikaros does not care how data is interpreted within modules after all.

VIII. VALIDATING MODELS

To automatically validate a model against relevant data, for example, neurobiological databases, the specification of a module can include the *models* attribute. For example, a module that claims to model the amygdala could be describes in the following way:

```
<module
   class = "MyClass"
   name = "MyModule"
   models = "Amygdala"
/>
```

This information could be used to match the graph made up of the modules in an Ikaros model to connectivity data found in neurobiological databases. Some first attempts towards such as system have been taken [11]. More recently, we also interfaced the Ikaros validation system with the CoCoMac database.

IX. EXPERIMENT DATABASE

In our earlier studies of classical conditioning we have developed an extensive database of the design and results of conditioning experiments. The development of this database started in 1996 and now contains approximately 200 different experiments. The database is stored in a way that allows the experimental descriptions to be used as input to computer simulations of learning by classical conditioning.

Unfortunately, this database was stored in a form that is not easy to access unless the previous simulator developed at LUCS is used. It also has the limitation that it only covers classical conditioning and not other learning paradigms. As a part of the Ikaros project, we want to extend the experiment database by adding more experiment types and by translating the database to a more accessible format.

In the future, we will add experiment description for other learning paradigms besides classical conditioning. This includes operant conditioning experiment as well as more cognitively oriented experiments. The goal is to cover all experiment types that are regularly used with animals and humans. We estimate that the final database will include approximately 1000 experiments.

The entry for each experiment will include all information that is necessary to reproduce the experimental conditions in a simulator or a real experiment. This includes detailed data of the stimuli used, the apparatus, the exact timing etc. It will be important to differentiate between the part of the experiment description that contains the logic of the experiment and features such as timing and spatial location that are often not essential. This will allow modelers to adapt experiments to their needs in much the same way that an experiment developed for one species has to be changed to fit another. The database will also contain experiment descriptions in narrative form and pointers to external databases such as Medline and BIOSIS when appropriate.

To allow easy access to the experiment database, it will be coded in the XML format that is widely used for online data. The choice of XML for the database is natural since it allows for an evolving and continually expanding database structure. It can also be used to mediate the transfer of information from other already existing databases. Apart from translating the already existing database to this format, we will also develop tools that can be used to encode and visualize experiments through a web-based interface.

X. DISCUSSION

During the last few years, Ikaros has been used to build a number of cognitive models and to control many different robots. This has to date resulted in over 40 scientific publications. For example, for cognitive modeling, it has been used in several models of cognitive development and the modeling developmental disorders [3], [7], plasticity in the somatosensory cortex [14] and to study different forms of learning [5] and emotion [20], [4]. A lot of the work on Ikaros has involved visual processing, for example models of visual contour processing [19] and models of visual attention [1], [2].

We have used Ikaros to control a number of different robotic hands built at Lund University Cognitive Science to investigate haptic perception [15], [16], [17], [18]. The hands have different sensors and different degrees of freedom and are all controlled by different neural network based architectures. In another line of research, we have looked at anticipation and navigation in mobile robots including the e-puck and the BoeBot [12], [13]. Here, Ikaros is used to implement very different models that are more classical in the sense that they use potential fields or planning approaches.

The approach in Ikaros to be minimally demanding regarding the types of architectures that can be built and the types of programming styles that can be used has proved to be very successful. It is also clear that many of the design choices made initially were sound and has contributed to the usefulness of the system. Unlike most other frameworks, Ikaros do not force the user into one theoretical model or into using extensive libraries even though such support is offered. This has made it easy for users of diverse backgrounds to quickly learn to use the system.

On the other hand, there are certain restrictions that limits for what systems Ikaros is useful. Some of these constraints certainly makes Ikaros less useful for some systems, in particular architectures that mainly relies on symbolic processing rather than numerical computation. We believe that for a tool to be useful, it is necessary that it is adapted for specific tasks and this inevitably makes it less useful for other tasks. For Ikaros, it was important that it could be used for real-time processing and for robot control, which makes it different from many other framework for more biologically based modeling. We also wanted Ikaros to run on almost any hardware which is the reason behind many of the design choices.

In summary, Ikaros has proven to be a very useful tool for building cognitive systems models and for robot control. It has evolved into a mature and stable system and has currently been adopted by several research groups within the cognitive sciences.

XI. ACKNOWLEDGEMENTS

We would like to thank all the people that have tested and commented on the system during its development, in particular Takashi Omori, Håkan Jonson, Kolbjörn Gripne, Lars Kopp, Chris Prince, Martin Butz, Stefan Karlsson, Stefan Winberg, Anders Karlström, Mikael Asker, Vin Thorsteinsdottir, Sigurbirna Haflidadottir, Kiril Kiryazov, Gianguglielmo Calvi. More information about Ikaros can be found at the project web site: http://www.ikaros-project.org.

REFERENCES

[1] C. Balkenius. Cognitive processes in contextual cueing. In F. Schmalhofer, R. M. Young, and G. Katz, editors, *Proceedings of the European Cognitive Science Conference 2003*, pages 43–47. Lawrence Erlbaum Associates, Mahwah, NJ, 2003.

[2] C. Balkenius, K. Åström, and A. P. Eriksson. Learning in visual attention. In *ICPR '04 workshop on learning for adaptable visual systems (LAVS)*. 2004.

[3] C. Balkenius and P. Björne. Toward a robot model of attention-deficit hyperactivity disorder (adhd). In C. Balkenius, J. Zlatev, H. Kozima, K. Dautenhahn, and C. Breazeal, editors, *Proceedings of the First International Workshop on Epigenetic Robotics: Modeling Cognitive Development in Robotic Systems*, volume 85 of *Lund University Cognitive Studies*. 2001.

[4] C. Balkenius and J. Morén. Emotional learning: A computational model of the amygdala. *Cybernetics and Systems*, 32(6):611–636, 2000.

[5] C. Balkenius and S. Winberg. Cognitive modeling with context sensitive reinforcement learning. In *Proceedings of AILS '04*. Dept. of Computer Science, Lund, 2004.

[6] Christian Balkenius, Birger Johansson, and Jan Moren. *Ikaros Control File Specification*. http://www.ikaros-project.org/2007/IKC10-20070601/, 2007.

[7] P. Björne and C. Balkenius. A model of attentional impairments in autism: First steps toward a computational theory. *Cognitive Systems Research*, 6(3):193–204, 2005.

[8] J. David Eisenberg. *SVG Essentials*. O'Reilly, 2002.

[9] David Flanagan. *JavaScript: the definitive guide*. O'Reilly, fourth edition, 2002.

[10] Bill O. Gallmeister. *POSIX.4—programming for the real world*. O'Reilly, 1995.

[11] M. Gustafsson and C. Balkenius. Using semantic web techniques for validation of cognitive models against neuroscientific data. In *Proceedings of AILS '04*. Dept. of Computer Science, Lund, 2004.

[12] B. Johansson. Elastic template matching in outdoor environments. Master's thesis, Lund Univeristy Cognitive Science, Lund, 2004.

[13] B. Johansson and C. Balkenius. An experimental study of anticipation in simple robot navigation. In M. et al Butz, editor, *Anticipatory Behavior in Adaptive Learning Systems: From Brains to Individual and Social Behavior*. Springer, 2007.

[14] M. Johnsson. Cortical plasticity: A model of somatosensory cortex. Master's thesis, Lund Univeristy Cognitive Science, 2004.

[15] M. Johnsson and C. Balkenius. Experiments with artificial haptic perception in a robotic hand. *Journal of Intelligent and Fuzzy Systems*, 17(4):377–385, 2006.

[16] M. Johnsson and C. Balkenius. LUCS haptic hand II. Technical Report 9, LUCS Minor, 2006.

[17] M. Johnsson and C. Balkenius. Neural network models of haptic shape perception. *Robotics and Autonomous System*, 22:720–727, 2007.

[18] M. Johnsson and C. Balkenius. Associating som representations of haptic submodalities. In *Proceedings of TAROS 2008*. Edinburgh, UK, 2008.

[19] Stefan Karlsson. Monocular depth from occluding edges. Master's thesis, Department of Mathematics, Lund Institute of Technology, 2004.

[20] J. Morén. *Emotion and Learning - A Computational Model of the Amygdala*. Lund University Cognitive Studies, 2002.

[21] Bradford Nichols, Bick Buttlar, and Jackie Proulx Farrell. *Pthreads Programming*. O'Reilly, 1996.

Multirobot Applications with the ThinkingCap-II Java Framework

H. Matínez-Barberá
Dept. of Information and Communications Engineering
University of Murcia
30100 Murcia, Spain
humberto@um.es

D. Herrero-Pérez
Dept. of Systems and Automation Engineering
University Carlos III of Madrid
28911 Madrid, Spain
dherrero@ing.uc3m.es

***Abstract*—We present a Java framework, ThinkingCap-II, for developing mobile multi-robot applications, which has been successfully used in indoor, automotive and industrial robotics applications. It consists on a reference cognitive architecture that serves as a guide for making the functional decomposition of a robotics system, a software architecture that allows a uniform and reusable way of organising software components for robotics applications, and a communication infrastructure that allows software modules to communicate in a common way. A key aspect of this software architecture is that it allows code reusability by high level abstraction and a uniform way of accessing the characteristics of the sensors. In order to show the suitability of the framework an autonomous vehicle case study is discussed.**

I. Introduction

The development of complex robotics applications involves diverse areas with different needs, such as data acquisition, signal processing, intelligent control, networking, etc. Large robotics projects involving developer teams require the efficient collaboration of their members, and also the easy integration of the individual developments. In the case of small or reduced development groups, without considering the economic factor, this becomes even more critical because the development can be extended during non-bearable periods, being of paramount importance fast prototyping and code reusability. To allow for these, the abstraction, organisation, and design of the software components are mandatory. In addition, it is also mandatory to produce working, robust and reliable applications.

The software aspects of this issue have not been discussed in-depth by the robotics community [1], probably because it is traditionally a software engineering topic. Nevertheless, the need of standard specifications that deal with the recurrent concepts and requirements of the robot software development is certainly a key issue, which would allow to share, distribute and/or reuse robotics software components. Thus, several recent papers in journals address robotics software surveys, analysis and comparisons (for instance see [2], [3]).

Traditionally, robotics software was basically the implementation of a functional architecture which was focused for a specific problem of set of problems. In this cases, the software was divided into modules depending of their functionalities (like *TCA* [4], *AuRA* [5], *3T* [6], and *Saphira* [7]), most of them obviating the transparency in communications, portability and code reusability. These usually group the data acquisition, the real-time reactive processing, and the computation of actuators to perform certain actions in a single software module, mainly due to the real-time constrain of these applications. On the other hand, robotics frameworks focused on low-level problems aims to provide an abstraction of the robotics platform (like *Player/Stage* [8], *Open-R* [9], and *OROCOS* [10]), which facilitates the reusability of software upon this abstraction. From the software point of view, it would be quite useful to define a generic robot and the separation of the problems to address. Then it would be possible to build tools to allow for productivity in the robotics development cycle (like *MissionLab* [11], *URBI* [12]). In the last years there is also an increasing trend in multi-platform support, either in the development of the whole software framework (i.e. *TeamBots* [13] written entirely in Java) or allowing clients developed in other programming languages, be then interpreted, scripted or compiled (i.e. both *Player/Stage* and *URBI* support remote clients written in different programming languages like Java, Python, etc).

When faced with the development of robotics applications for different domains, platforms, sensors and actuators using a reduced development team, like our research group is, **productivity** is of paramount importance. In addition, if any of the developments is to become a commercial or industrial product, the **robustness** is also a must. We have summarised important properties of selected frameworks regarding the productivity and robustness goals in Table. I. The *Research* column identifies the most relevant users base in the research community. The *Industrial* column identifies which frameworks are intended to be used for commercial or industrial applications, and what degree of industrial grade has been reached. The *Prototyping* column evaluates the simplicity or easiness for fast prototyping, which is directly related to man-hours effort. The *Language* column shows the language for implementing and using the framework. In some cases, there are available clients for additional languages. The *Data Flow* column identifies if the data flow is fixed at compilation time or can be configured at runtime. The *Functional* column identifies which kind of functional architecture the frameworks are related to, if any.

Because we are concerned on productivity, fast prototyping is a property that is considered more than necessary. *URBI*,

	Research	Industrial	Prototyping	Language	Data Flow	Functional
Player	Widespread			C++/Clients		
URBI	Accademia	Limited	Excellent	URBI/Clients	Fixed	FSM
Open-R	RoboCup	Full		C++	Configurable	
OROCOS	EU Project	Full		C++	Configurable	
TeamBots	Limited		Excellent	Java	Fixed	Reactive
MissionLab	Military	Full	Adequate	C++	Fixed	AuRa

TABLE I
SUMMARY OF SELECTED FRAMEWORKS PROPERTIES

TeamBots and *MissionLab* provide good support for this, but only *MissionLab* and *URBI* can be qualified as industrial grade. On the other hand, general usability is assured by *Player/Stage*, *Open-R* and *OROCOS*, but only at a platform level. In addition, Java multi-platform development has become a standard feature in business, with many productivity and development tools readily available. For this reason, we have developed a Java software framework for robotics applications that tries to keep productivity and robustness as it main goals, while adopting the many interesting features of the above mentioned frameworks.

This paper presents a software framework for developing autonomous robots applications in diverse domains, like laboratory robots, automotive and industrial vehicles are. The main goal of the framework is to allow a high productivity while obtaining robust code, adequate for commercial or industrial applications. The Java framework has been successfully used in different applications like laboratory robots, soccer-playing robots, industrial robots and autonomous vehicles.

The paper is organized as follows. The first section describes the characteristics and design criteria of the software robotics framework which is later used to control very different platforms. The second section analyses and discusses the most important features of the proposed framework. The third section describes two case-studies: an autonomous car and an industrial mobile robot. Finally, some conclusions are presented.

II. THE THINKINGCAP-II FRAMEWORK

ThinkingCap-II (TC-II) is a Java framework for developing mobile robot applications[1]. It is a joint effort between the University of Murcia, Spain, and the University of Örebro, Sweden, and it is based on previous work on *ThinkingCap* [14], [15] and *BGA* [16] architectures. The framework consists of a reference cognitive architecture (largely based on *ThinkingCap*) that serves as a guide for making the functional decomposition of a robotics system, a software architecture (partially based on *BGA*) that allows a uniform and reusable way of organising software components for robotics applications, and a communication infrastructure that allows software modules to communicate in a common way, independently of whether they are local or remote.

A. Functional architecture

Although the *TC-II* framework is functional architecture-free, we have developed most of our applications (like the the case study described below) following a functional architecture based on *ThinkingCap* [15]. It consists of a two-layer architecture (Fig. 1) for controlling mobile robots, one layer for reactive processes and the other for deliberative processes. It can be viewed as a stripped down instance of a *3T* architecture. The modules group the different functionalities present in typical mobile robotics systems (navigation, perception, control and planning), in which sensing and acting are a must. An important role is played by a centralised data structure called *Local Perceptual Space* (LPS), borrowed from the *Saphira* architecture [7]. It is a geometrically consistent robot centric space which consists of a collection of *Local Perceptual Objects* (LPOs). These LPOs model the local environment of the robot, and take into account the a priori information (map) and the currently perceived information (sensors) in a coherent way.

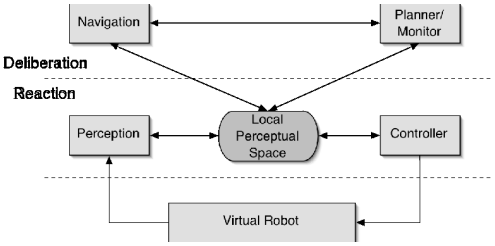


Fig. 1. ThinkingCap-II functional architecture

A key point of the architecture is the *VirtualRobot* module, which provides an abstract interface to the sensori-motoric functionalities of the robot, effectively hiding the hardware components, much like *Open-R*, *Player/Stage*, and *OROCOS* do, but at a higher level. In this sense, we have developed *VirtualRobot* modules that run on top of *Open-R* for controlling AIBOs and on top of *Player/Stage* for controlling Pioneers.

This architecture has been implemented and used in different types of robots and has shown good capabilities as an abstract guideline to organise the software which has to be run in a robot.

B. Software architecture

The framework defines an abstract model of a *TC-II* module, which all the modules must follow. Some of these modules will correspond to modules of the functional architecture. Depending on the complexity of the system there could be one to one or one to many correspondences. For instance the *Perception* can be implemented as a single module or as a collection of sub-modules, but in either case the modules must

[1]Additional information can be found at http://robolab.inf.um.es/tc2

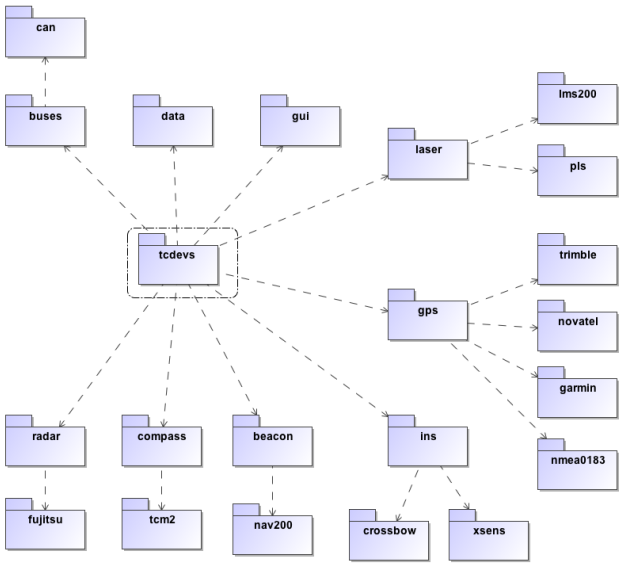


Fig. 2. Package *tcdevs*

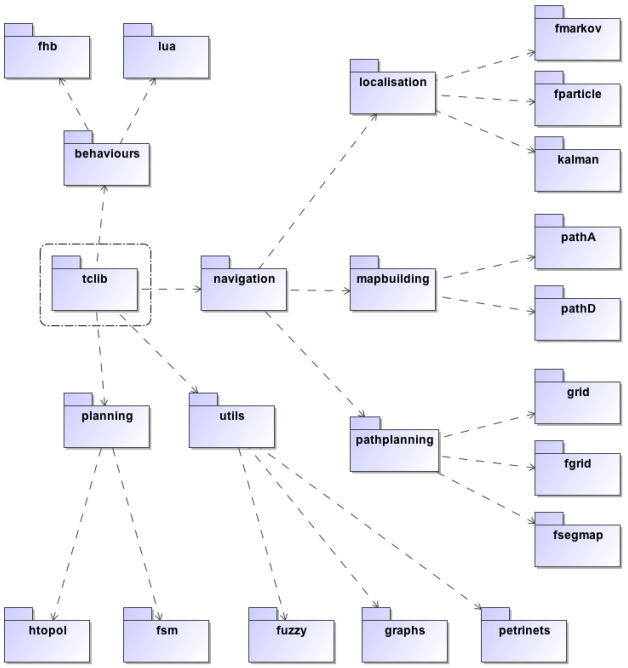


Fig. 3. Package *tclib*

stick to the abstract module definition, which has support for single thread and multi thread execution.

As the modules can be distributed among a set of CPUs, the framework relies on a centralised communication scheme, where all the communication goes through a blackboard (described in section II-C). In addition, this blackboard is based on an event system so that modules do not need to poll the blackboard but wait until the desired type of event has occurred. The communication mechanism is detailed in the next section. The abstract *TC-II* module includes a port to put data into the blackboard and to receive data from the blackboard (via both polling and events).

The software architecture makes extensive use of all the Object Oriented features of the Java language, and it includes and offers: specification of run-time parameters for the different modules, flexible configuration of the system (in terms of modules, robots and CPUs), and a predefined components library. These features are organised in different packages:

- ***tcdevs***. This package deals with device communication and data representation issues, both for sensors and actuators (Fig. 2). It includes a broad range of sensor types (laser, GPS, INS, radar, etc) and makes extensive use of the Factory pattern to hide the implementation of their device drivers, allowing the addition of additional vendor specific drivers in a transparent way, which is resolved at run time. The most important benefit of this is that the robot code does not know about and is independent of the actual devices, allowing for a high degree of reusability. In addition, the package includes implementations to access different data buses, like CAN, I2C, RS-232, etc.
- ***tclib***. This package is a repository of general algorithmic solutions to standard mobile robotics problems (Fig. 3). It includes code for localisation (fuzzy-Markov filter, particle filter, Kalman filter), map-building (grid, fuzzy grid, fuzzy segments), path-planning (A*, D*), task-planning (topological planning, finite state machine, Petri net), and behaviour execution (fuzzy hierarchical behaviours, LUA embedded behaviours). The idea of this package is to allow for new developments to be fast prototyped by reusing previously used solutions. The designer has simply to instantiate any of these methods from inside the desired module. The included techniques make use of the Model-View-Control pattern to allow for a general implementation of the algorithms which is independent of the visualisation of the results. Most techniques include a default visualisation module which greatly simplifies the debugging process.
- ***tcarch***. This package deals with architectural issues, high level communications and modularisation (Fig. 4). On the one hand, it includes the implementation of the Linda communication infrastructure (described in section II-C), basic runtime operation (thread and event management), and shared data types (LPS, LPO and world models). On the other hand, it includes basic abstract implementations of the different modules of the reference functional architecture (*VirtualRobot*, *Controller*, *Perception*, *Navigation*, *Planner*, *Monitor*), which are then customised for any given application by sub-classing.

One key aspect of the framework is that the run-time characteristics of the system can be specified and customised by the use of configuration files. The framework supports two different types of general configuration files, and contains methods to parse and verify them. The following configurations are used by all the modules:

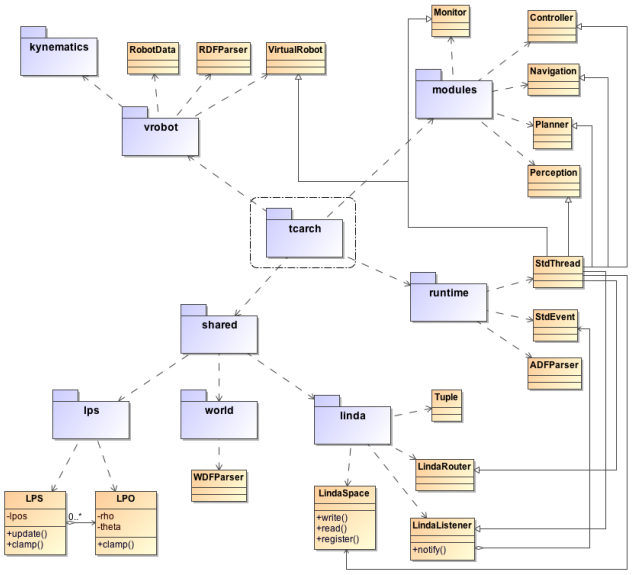


Fig. 4. Package *tcarch*

- *Architecture Definition File* (ADF). Specifies which modules are to be run, on which CPUs they will be running, and which type of communication and process synchronisation mechanism they will be using. The framework provides methods to automatically instantiate the corresponding classes at run-time. Thus, the use of ADF allows for great flexibility when trying different approaches and provides a very convenient method for specifying a distributed system.
- *Robot Description File* (RDF). Specifies the different parameters related to a given robot, like sensor number, types, position and orientation, and platform kinematics model and its parameters. It includes the specification of the actual classes that will handle device-specific issues, implementing interfaces defined in the *tcdevs* package. The framework provides methods to access the definition of the robot, and also to display it.

In addition to these, the framework also includes an application specific configuration file, which may or may not be used by the different modules:

- *World Description File* (WDF). Specifies the a priori knowledge of the robot environment, like walls, rooms, corridors, landmarks, areas, waypoints, etc. The framework provides methods to access and display this information. The WDF can be left empty if no a priori information exists. The *TC-II* framework includes a graphical tool for generating WDFs.

In order to execute a *TC-II* based robot, a valid ADF and a unique name are needed. The name is used to identify the robot should more than one is used. The framework parses the ADF and then loads and instantiates all the different modules specified, each with its corresponding parameters and the desired Linda events that it will be listening to. All the modules of a single robot connect to the robot local blackboard. In a

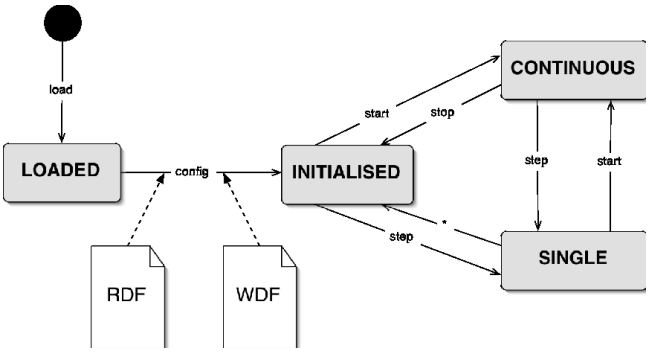


Fig. 5. Runtime module's finite state machine

multi-robot scenario, the ADF also contains the address of the global blackboard. Once a module is instantiated, it execution state is controlled by a finite state machine (Fig. 5), which can in one of the following states: LOADED (it have just been instantiated), INITIALISED (a valid RDF and WDF has been received from the blackboard, and the module is ready for operation), CONTINUOUS (for continuous execution) or STEP (for step by step execution for debugging purposes).

C. Communication support

In a distributed system, as many robotics applications are, information sharing is a key point. Both the functional and software architectures allow the execution of modules in different processes or machines. To allow for a flexible information exchange mechanism, *TC-II* relies on a shared blackboard with similar services to those offered by a Linda system [17]. As in typical blackboard systems, each module reads information from the blackboard, processes it, and then writes the corresponding results. Besides this, it can also work as an event driven blackboard. In this way, each module registers into the blackboard which kind of data it desires to receive. When new data of such type is available it is sent directly to the module.

Shared information is exchanged using tuples, a tuple being a pair <*key*, *item*>. *Key* identifies the kind of data, and *item* is the actual data. For instance there are some tuple types for different purposes, like sensor data, motion commands, debug information, navigation data and others. The object-oriented capabilities of the Java language are used in this case to define and implement all these tuples from an object-oriented paradigm.

The blackboard of *TC-II* has been designed to work in distributed scenarios, with different modules running in different machines. In a multi-robot scenario, each robot maintains its own local blackboard, and in addition there is a global blackboard, which is an extension of the local blackboard concept. In this case, information is now exchanged using triples, a triple being <*id*, *key*, *item*>, where *id* uniquely identifies the robot that produced the information. The framework includes a specialised module *LindaRouter* which connects each local blackboard with the global one. This module is in charge of filtering information sent to and received from the

global blackboard, thus acting as a lightweight communication router.

III. Framework Discussion

A. On the use of Java

The *TC-II* framework has been fully implemented in Java. One of the advantages of using Java is the real achievement of having platform independence. Thus, the development of the components is typically done in desktop computers with standard operating systems, while the actual deployment occurs in embedded systems that support Java. The current, tested, deployment systems include both Linux and Linux RT and embedded Java microcontrollers. Moreover, the Java implementation makes the integration of the human-interfaces in distributed scenarios a simple process. Being an interpreted language, it is not suitable for computationally intensive tasks, like real-time vision is. Because of this, the Java Native Interface (JNI) provides a handy way of calling C-based software by the virtual machine.

Some of the virtues of Java for development are related to robustness (Java programs run in a protected virtual machine, Java always performs runtime bounds checking) and reduced debugging time (parameters always passed by value, except objects that are accessed by reference, automatic garbage collection, rigid type safety except for widening conversions). Robotics applications may benefit from the productivity associated, which is habitual in many corporative and enterprise software developments.

An important feature of many robotics frameworks is the ability of specifying the data flow at run-time. As such, *Open-R* and *OROCOS* allow such possibility, in the first case by means of configuration files. In *TC-II*, this is accomplished by way of the ADFs, because they not only contain which modules are to be active at runtime, but also which events are to receive. The way this is actually implemented is by registering a given module into the LindaSpace to receive a set of tuple *keys*. When a new value for a given *key* is available, the blackboard calls a *notify()* method of the registered module with the corresponding tuple. By writing the appropriate code in the *notify()* method, a different variety of data flows can be accomplished. This registration occurs at runtime when parsing some fields of the ADF for the corresponding module. This is possible by making extensive use of Java's reflection properties and Factory patterns.

Another important aspect is code reusability. The framework allows the developer to write robot independent code through the use of RDFs. For instance if a developer writes a perception routine that computes some feature depending on the sensor configuration, the RDFs allow for a high level of abstraction and a uniform way of accessing the characteristics of the sensors. Then the system designer or integrator has to provide only the number and location of the sensors to use the routine. All the standard components library has been written following this approach. This is also possible by making extensive use of Java's reflection properties and Factory patterns.

A typical concern with Java is performance. While early versions of Java were significantly outperformed by statically compiled languages such as C++, and it may still be the case for embedded systems because of the requirement for a small footprint, current just in time (JIT) compiler technology are closing the performance gap for long-running Java processes, like robotics applications are, were the classes used during the execution of the applications do not vary much during time. Most modern Java virtual machines support JIT compilation (as the one used in the iFork case study, section IV). One comprehensive study of microbenchmarks [18] shows quite a large variation in results but indicates that Java often outperforms C++ in operations such as memory allocation and file I/O while C++ often outperforms Java in arithmetic and trigonometric operations.

B. On the functional architecture

We can distinguish between frameworks that provide tools to design customised systems, and the frameworks that force the use of specific functional architectures of the control software. The first group includes *Player*, *URBI*, *OPEN-R*, *OROCOS*, and *TC-II* while *TeamBots* and *MissionLab* belong to the second group.

The first group provides a generic abstraction of the hardware layer, which is quite useful to separate the robot control and the real-time control of effectors. The different levels of abstraction depend on the complexity of the hardware platform for which they are designed. *Player* is intended to command wheeled robots. *URBI* is designated to command more generic platforms and although complex commands can be written, its kernel is low level in essence. *OROCOS* propose to define a generic robot by the specification of components, which are completely decoupled of communications, and hence, control flow and data flow are established outside of components. The idea is that the user be able to assemble and achieve the global functionality of the robot. The components used by the application are chosen by the developer depending on the functionalities needed.

TC-II follows a similar approach to that of *OROCOS*, and, in essence, the software architecture is de-coupled from the functional architecture. How is this related with what it is stated in section II-A? Basically, the software architecture does not know anything about the current functional architecture, because the actual data flow is instanced at runtime, and hence the functional architecture is then established. On the other hand, our typical applications are layered up using the described functional architecture, and thus we provide specific classes to allow for a straightforward implementation of it, but nonetheless, it is neither mandatory nor necessary to follow it. In fact, one examples of the case studies presented below does not strictly follows the functional architecture (see section IV).

C. Simulation

Platform independence is guaranteed by the use of the Java language, and it is a very important feature of the *TC-II* framework. Platform independence allows running and

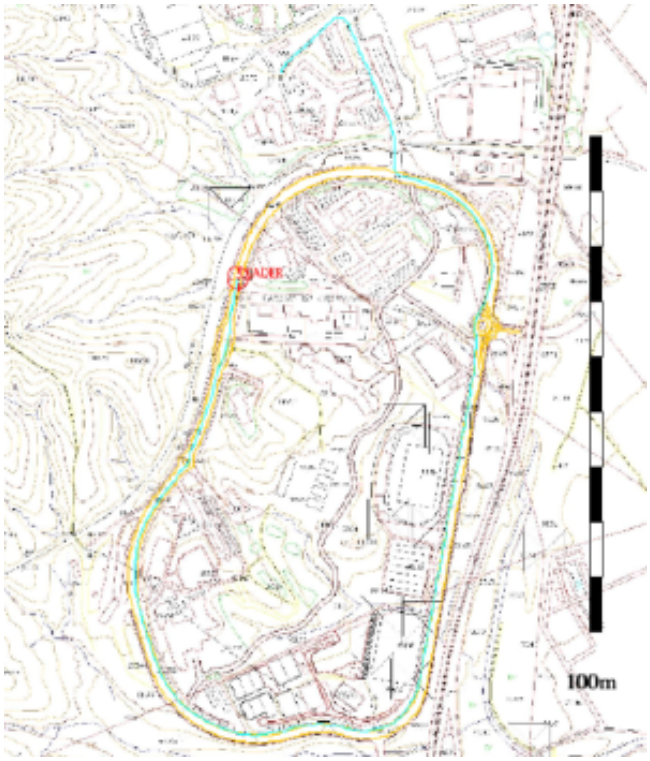


Fig. 6. ThinkingCap-II simulator

debugging the code in personal computers and testing it in the real platform. The *TC-II* framework takes advantage of this and includes a *Simulator* module (Fig. 6) that can simulate different sensor types, like range sensors (sonar, laser, radar) and positioning sensors (GPS, laser, compass), and platform models, like differential, tricycle, leg, and Ackerman drives. The sensor simulation is realistic enough for taking into account multi-path reflection, noise, and different firing patterns, while the platform simulation is based on kinematics equations and some pseudo dynamics constraints (i.e. the minimum time to perform a full turn of the steering wheel). The combination allows for testing of the efficiency and performance of the different modules and their algorithms with an acceptable degree of realism. In addition, the simulator can simulate multiple robots, and their sensors not only reflect the environment but also the other robots.

The *Simulator* is implemented as a *VirtualRobot* that is not attached to any real device, and the same RDFs used by the real robots are used by the simulator to configure the sensor types and models and the platforms kinematics and constraints. Thus, switching between a real robot and a simulated one is as simple as changing a class name of the *VirtualRobot* section in the ADF. In addition, the model of the environment is specified using a WDF (which may be the same as that used by the robots).

IV. Case-Study: Intelligent Vehicles

This application is part of the MIMICS project, which aims to develop an intelligent platoon of vehicles [19], where the

Fig. 7. The SatAnt autonomous car

leading vehicle (which is manned) acts as a guide for the following vehicles (which are unmanned)[2]. Because of limited budget, only one autonomous car has been developed and built. The operation of the leading car is quite simple: it uses its sensors to send information to the following car, which then uses both its sensors and the information received to control the actuators. All the information is shared using wireless links.

The autonomous car, called SatAnt (Fig. 7), is based on a COMARTH S1-50 sport car, which has been heavily modified to allow it to be controlled by a computer based system. The modifications include an automatic gearbox, electronic assisted steering system, electronic speed control, and electronic braking system. For safety reasons, all electronic systems have been designed in such way that they allow both manual and automatic control, and at any time the electronic systems can be disengaged. Both the frame and the outer shell have been modified to accommodate for the non-standard equipment. The sensors system includes a Novatel GPS (which provides global positioning data), a Precision Navigation electronic compass (which provides both heading and pitch/roll data), relative encoders attached to the four wheels (which provide vehicle speed), absolute encoder attached to the steering wheels arm (which provides steering wheels angle), and a Fujitsu 77 GHz radar (for detecting obstacles and the leading car). The manned car is simply provided with portable equipment that contains the positioning sensors, a small processing unit and the radio communication link. This portable system can be used in any standard car.

A typical instance of the MIMICS application consists of (Fig. 8): one manned car, one SatAnt unmanned car, and an operator base station. The base station holds the instance of the global blackboard and a GIS application to monitor the state of the system. The manned car holds a local blackboard with has attached a *VirtualRobot* and a *Peception* module which integrates sensor information (GPS position, electronic compass heading and, optionally, pulses from the tachometer) into

[2]Details and videos can be found at http://robolab.inf.um.es/mimics

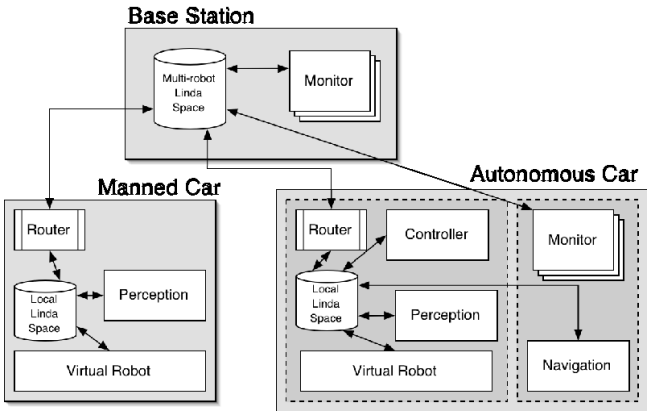


Fig. 8. The MIMICS application architecture

a Kalman filtered position estimation, which is then used as a reference by the unmanned car. The unmanned car architecture is more complex because processing is distributed into two CPUs: one for time-critical modules (the reactive layer) and the other for non time-critical modules (the deliberative layer and the user interface), as depicted in Fig. 8. These modules are:

- *VirtualRobot* reads sensor information (GPS position, electronic compass heading, tachometer pulses from the four wheels, and radar targets) and writes it into the local Linda Space. It also reads control values from the Linda Space and sends them to the corresponding actuators. Both low level controllers and sensors are interfaced by a CAN bus, which connects the *VirtualRobot* to a series of Java programmed micro-controllers that perform velocity control, steering wheel control and brake control.
- *Perception* reads the raw positioning data and applies a Kalman filter to fuse them and produce a corrected global position, which is then written into the Linda Space. Moreover, the raw radar targets are stored in a radar buffer, from which an estimate of the time for collision is computed, and false targets are filtered out. This estimate is also written into the Linda Space.
- *Controller* reads the positioning data, collision data, and desired path from the Linda Space, executes the different reactive behaviours, and then produces the corresponding control values, which are written into the Linda Space.
- *Navigation* reads the position of the leading car from the Linda Space and generates a path to guide the vehicle (connecting the leading car positions) and the desired velocity (averaging the leading car speed over a period). Both the path and the velocity are written into the Linda Space.
- *Monitor* simply displays the position of the different vehicles and their trajectories on a map, and also allows the operator to take control over an autonomous car and teleoperate it using a joystick.

Being a complex and distributed application, it has benefited from the *TC-II* architecture in a number of ways. The runtime characteristics of the system allowed for a fast development of the first prototype (9 man/months for the software compared to 21 man/months for the hardware), in which only one full scale test was conducted (it required booking a private car race track, bringing the two cars, laboratory equipment and a five persons team some 80 km from our lab). This was possible due to the implementation in Java and the use of ADFs, which allowed us to try different pieces of the system in single desktop computers, single embedded computers, or combinations of desktop and embedded computers. In addition we reused algorithms and device drivers that were developed for other projects.

V. Conclusions

This paper has shown *ThinkingCap-II*, a Java framework for mobile robotics applications, which has been designed with to main goals into mind: productivity and robustness. These are achieved by a combination of a methodology to decompose a robotics system into different modules based on functionality, a software architecture to provide run-time support, dynamic configuration and a components library, and a communication infrastructure that allows distribution of the different components. This framework has been successfully used in indoor robotics, automotive and industrial applications, where the physical platforms are quite different.

An application developed using the proposed framework has been described and commented: an autonomous vehicle. It has benefited of the properties of the framework, as fast prototyping, distributed nature and multi-platform are. Thus we have also shown that Java can be used for real robotics applications, even in demanding environments like the transportation one is. Most concerns about Java performance are clarified by the use of modern JIT technologies, which only add a small processing overhead compared to natively compiled code. In addition, using Java for the high-level framework has the advantage of allowing a faster design and development cycle. In addition, all the high level software can be effectively tested in any platform before the actual deployment of the code. The advantages of using the approach presented are very obvious for small research and development teams.

Acknowledgments

This work has been supported by CICYT projects TIC2001-0245-C02-01, DPI-2004-07993-C03-02 and DPI-2007-66556-C03-02 and PROFIT projects FIT-1602000-2001-53, FIT-160300-2002-82, FIT-160300-2003-41, Spanish Ministry of Science and Innovation. Special thanks to Alessandro Saffiotti for his valuable help and comments.

References

[1] J. Fernández-Madrigal, C. Galindo, J. González, E. Cruz-Martín, and A. Cruz-Martín, "A software engineering approach for the development of heterogeneous robotic applications," *Robotics and Computer-Integrated Manufacturing*, vol. 24, no. 1, pp. 150–166, 2008.

[2] A. Oreback and H. Christensen, "Evaluation of Architectures for Mobile Robotics," *Autonomous Robot*, vol. 14, pp. 33–49, January 2003.

[3] J. Kramer and M. Scheutz, "Development environments for autonomous mobile robots: A survey," *Autonomous Robots*, vol. 22, no. 2, pp. 101–132, 2007.

[4] R. Simmons, "Structured Control for Autonomous Robots," *IEEE Transactions on Robotics and Automation*, vol. 10, no. 1, pp. 34–43, 1994.
[5] R. Arkin and T. Balch, "AuRA: Principles and Practice in Review," *Journal of Experimental and Theoretical Artificial Intelligence (JETAI)*, vol. 9, no. 2/3, pp. 175–188, 1997.
[6] R. Bonasso, R. Firby, E. Gat, D. Kortenkamp, D. Miller, and M. Slack, "Experiences with an Architecture for Intelligent, Reactive Agents," *Journal of Experimental and Theoretical Artificial Intelligence*, vol. 9, no. 2/3, pp. 237–256, 1997.
[7] K. Konolige, K. Myers, E. Ruspini, and A. Saffiotti, "The Saphira Architecture: A Design for Autonomy," *Journal of Experimental and Theoretical Artificial Intelligence (JETAI)*, vol. 9, no. 1, pp. 215–235, 1997.
[8] B. Gerkey, R. Vaughan, and A. Howard, "The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems," in *International Conference on Advanced Robotics (ICAR)*, Coimbra, Portugal, July 2003, pp. 317–323.
[9] M. Fujita and K. Kageyama, "An Open Architecture for Robot Entertainment," in *International Conference on Autonomous Agents*, Marina del Rey, California, United States, 1997, pp. 435–442.
[10] H. Bruyninckx, "Open robot control software: the OROCOS project," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2001, pp. 2523–2528.
[11] D. Mackenzie, R. Arkin, and J. Cameron, "Multiagent mission specification and execution," *Autonomous Robots*, vol. 4, no. 1, pp. 29–52, 1997.
[12] J. Baillie, "Urbi: towards a universal robotic low-level programming language," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2005, pp. 820–825.
[13] T. Balch. (2000) TeamBots. [Online]. Available: http://www.cs.cmu.edu/~trb/TeamBots/
[14] A. Saffiotti, K. Konolige, and E. Ruspini, "A Multivalued-Logic Approach to Integrating Planning and Control," *Artificial Intelligence*, vol. 76, no. 1-2, pp. 481–526, 1995.
[15] A. Saffiotti, "Autonomous Robot Navigation: a Fuzzy Logic Approach," Ph.D. dissertation, Universite Libre de Bruxelles, Belgium, 1998.
[16] H. Martínez-Barberá and A. Gómez-Skarmeta, "A Framework for Defining and Learning Fuzzy Behaviours for Autonomous Mobile Robots," *Intl. J. of Intelligent Systems*, vol. 17, no. 1, pp. 1–20, 2002.
[17] D. Gelernter, "Generative Communication in Linda," *ACM Trans. on Programming Languages and Systems*, vol. 17, no. 1, pp. 80–112, 1985.
[18] T. Bruckschlegel, "Microbenchmarking $C++$, $C\sharp$ and Java," *Dr. Dobbs*, June 2005.
[19] A. Gómez-Skármeta, H. Martínez-Barberá, M. Zamora, B. Úbeda, F. Gómez, and L. Tomás, "Mimics: exploiting satellite technology for an intelligent convoy," *IEEE Intelligent Systems*, vol. 17, no. 4, pp. 85–89, 2002.

Incremental Component-Based Construction and Verification of a Robotic System

Ananda Basu, Matthieu Gallien, Charles Lesire, Thanh-Hung Nguyen,
Saddek Bensalem, Félix Ingrand and Joseph Sifakis

***Abstract*— Autonomous robots are complex systems that require the interaction/cooperation of numerous heterogeneous software components. Nowadays, robots are critical systems and must meet safety properties including in particular temporal and real-time constraints. We present a methodology for modeling and analyzing a robotic system using the BIP component framework integrated with an existing framework and architecture, the LAAS Architecture for Autonomous System, based on GenoM. The BIP componentization approach has been successfully used in other domains. In this study, we show how it can be seamlessly integrated in the preexisting methodology. We present the componentization of the functional level of a robot, the synthesis of an execution controller as well as validation techniques for checking essential "safety" properties.**

I. Introduction

A central idea in systems engineering is that complex systems are built by assembling components (building blocks). Components are systems characterized by an abstraction that is adequate for composition and re-use. It is possible to obtain large components by composing simpler ones. Component-based design confers many advantages such as reuse of solutions, modular analysis and validation, reconfigurability, controllability, etc.

Autonomous robots are complex systems that require the interaction/cooperation of numerous heterogeneous software components. They are critical systems as they must meet safety properties including in particular, temporal and real-time constraints.

Component-based design relies on the separation between coordination and computation. Systems are built from units processing sequential code insulated from concurrent execution issues. The isolation of coordination mechanisms allows a global treatment and analysis.

One of the main limitations of the current state-of-the-art is the lack of a unified paradigm for describing and analyzing the information flow between components. Such a paradigm would allow system designers and implementers to formulate their solutions in terms of tangible, well-founded and organized concepts instead of using dispersed coordination mechanisms such as semaphores, monitors, message passing, remote call, protocols, etc. It would allow in particular, a comparison of otherwise unrelated architectural solutions and could be a basis for evaluating them and deriving implementations in terms of specific coordination mechanisms.

A. Basu, T.-H. Nguyen, S. Bensalem and J. Sifakis are with VERIMAG CNRS/University Joseph Fourier, Grenoble, France.

M. Gallien, C. Lesire and F. Ingrand are LAAS/CNRS, Unversity of Toulouse, Toulouse, France.

The designers of complex systems such as autonomous robots need scalable analysis techniques to guaranteeing essential properties such as the one mentioned above. To cope with complexity, these techniques are applied to component-based descriptions of the system. Global properties are enforced by construction or can be inferred from component properties. Furthermore, componentized descriptions provide a basis for reconfiguration and evolution.

We present an incremental componentization methodology and technique which seamlessly integrate with the already existing LAAS architecture for autonomous robot. The methodology considers that the global system architecture can be obtained as the hierarchical composition of larger components from a small set of classes of *atomic* components. Atomic components are units processing sequential code that offer interactions through their interface. The technique is based on the use of the *Behavior-Interaction-Priority* (BIP) [2] component framework which encompasses incremental composition of heterogeneous real-time components.

The main contributions of the paper include:

- A methodology for componentizing and architecting autonomous robot systems applied to the existing LAAS architecture.
- Composition techniques for organizing and enforcing complex event-based interaction using the BIP framework.
- Validation techniques for checking essential properties, including scalable compositional techniques relying on the analysis of the interactions between components.

The paper is structured as follows. In Section II we illustrate with a real example, the preexisting architecture (based on GenoM [6]) of an autonomous robotic software developed at LAAS. From this architecture, we identify the atomic components used for the componentization of the robot software in BIP. Section III provides a succinct description of the BIP component framework. Section IV presents a methodology for building the BIP model of existing GenoM functional modules and their integration with the rest of the software. Controller synthesis results as well as "safety" properties analysis are also presented. Section V concludes the paper with a state of the art, an analysis of the current results and future work directions.

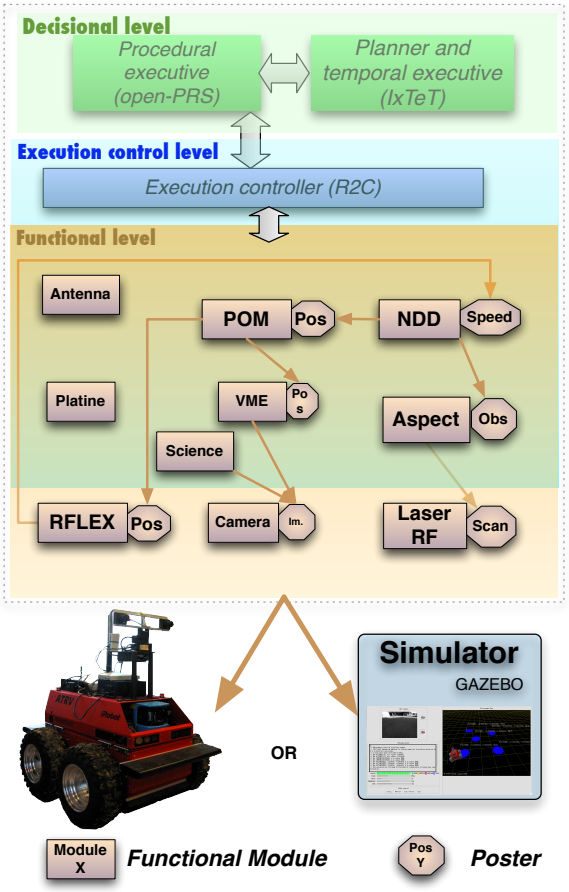


Fig. 1. An instance of the LAAS architecture for the DALA Robot.

II. Modular Architecture for Autonomous Systems

At LAAS, researchers have developed a framework, a global architecture, that enables the integration of processes with different temporal properties and different representations. This architecture decomposes the robot system into three main levels, having different temporal constraints and manipulating different data representations [1]. This architecture is used on a number of robots (e.g. DALA, an iRobot ATRV) and is shown on Fig. 1. The levels in this architecture are :

- a *functional* level: it includes all the basic built-in robot action and perception capacities. These processing functions and control loops (e.g., image processing, obstacle avoidance, motion control, etc.) are encapsulated into controllable communicating modules developed using GenoM[1]. Each modules provide *services* which can be activated by the decisional level according to the current tasks, and *posters* containing data produced by the module and for other (modules or the decisional level) to use.

[1]The GenoM tool can be freely downloaded from: `http://softs.laas.fr/openrobots/wiki/genom`

- a *decisional level*: this level includes the capacities of producing the task plan and supervising its execution, while being at the same time reactive to events from the functional level.
- At the interface between the decisional and the functional levels, lies an *execution control level* that controls the proper execution of the services according to safety constraints and rules, and prevents functional modules from unforeseen interactions leading to catastrophic outcomes. In recent years, we have used the R2C [14] to play this role, yet it was programmed on the top of existing functional modules, and controlling their services execution and interactions, but not the internal execution of the modules themselves.

The organization of the overall system in layers and the functional level in modules are definitely a plus with respect to the ease of integration and reusability. Yet, an architecture and some tools are not "enough" to warrant a sound and safe behavior of the overall system.

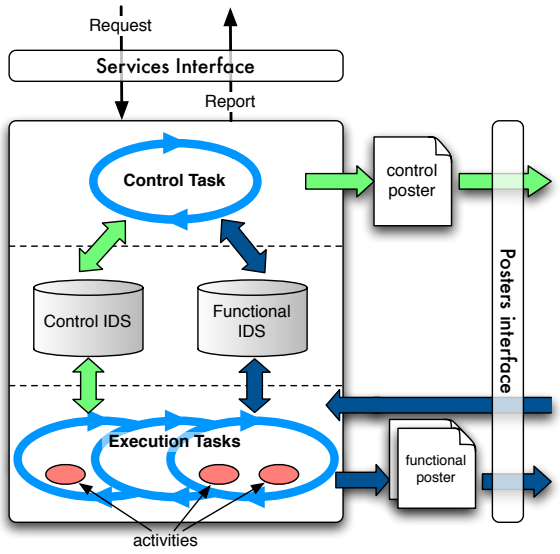


Fig. 2. A GenoM module organization.

In this paper the componentization method we propose will allow us to synthesize a controller for the overall execution of all the functional modules and will enforce by construction the constraints and the rules between the various functional modules. Hence, the ultimate goal of this work is to implement both the current *functional level* and *execution control level* with BIP.

A. GenoM Functional Modules

Each module of the LAAS architecture functional level is responsible for a function of the robot. Complex modalities (such as navigation) can be obtained by having modules "working" together. For example in Fig. 1 (which only shows the data flow of the functional level), there is an explicit periodical processing loop. The module Laser RF acquires the laser range finder and store them in the poster `Scan`,

from which Aspect builds the obstacles map `Obs`. The module NDD (responsible for the navigation) avoids these obstacles while periodically producing a `Speed` reference to reach a given target from the current position `Pos` produced by POM. Finally, this `Speed` reference is used by RFLEX, which controls the speed of the robots wheels, and also produces the odometry position to be used by POM to generate the current position.[2]

All these modules are built using a unique generic canvas (Fig. 2) which is then instantiated for a particular robot function.

Each *module* can execute several *services* started upon upper level requests. The module can send information relative to the executed requests to the client (such as the final report) or share data with other modules using *posters*. E.g. the NDD *module* provides six *services* corresponding to initializations of the navigation algorithm (*SetParams*, *SetDataSource* and*SetSpeed*), launching and stopping the path computation toward a given goal (*Stop* and *GoTo*) and a permanent service (*Permanent*). To execute this path, NDD exports the `Speed` poster which contains the speed reference.

The *services* are managed by a *control task* responsible for launching corresponding *activities* within *execution tasks*.

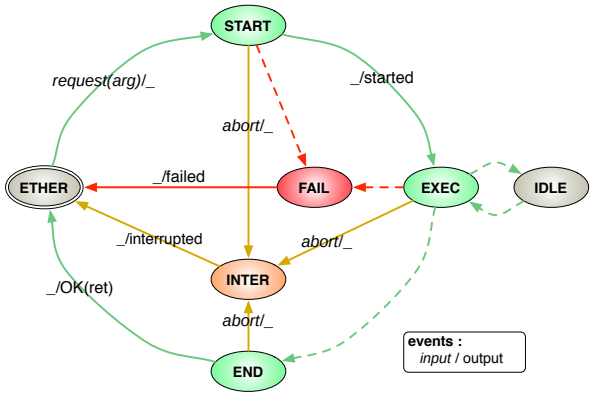


Fig. 3. Execution automaton of an activity.

Control and execution tasks share data using the internal data structures (IDS). Moreover execution tasks have periods in which the several associated *activities* are scheduled. It is not necessary to have fixed length periods if some services are aperiodic. Fig. 3 presents the automata of an *activity*. Activity states correspond to the execution of particular elementary code (codels) available through libraries and dedicated either to initialize some parameters (START state), to execute the activity (EXEC state) or to safely end the activity leading to reseting parameters, sending error signals, etc.

[2]This particular setup will serve as an example throughout the rest of the paper.

III. The BIP Component Framework

BIP[3] [2] is a software framework for modeling heterogeneous real-time components. The BIP component model is the superposition of three layers: the lower layer describes the *behavior* of a component as a set of *transitions* (*i.e* a finite state automaton extended with data); the intermediate layer includes *connectors* describing the *interactions* between transitions of the layer underneath; the upper layer consists of a set of *priority* rules used to describe scheduling policies for interactions. Such a layering offers a clear separation between component behavior and structure of a system (interactions and priorities).

BIP allows hierarchical construction of *compound* components from *atomic* ones by using connectors and priorities.

An *atomic* component consists of a set of *ports* used for the synchronization with other components, a set of transitions and a set of local variables. Transitions describe the behavior of the component. They are represented as a labeled relation between *control states*.

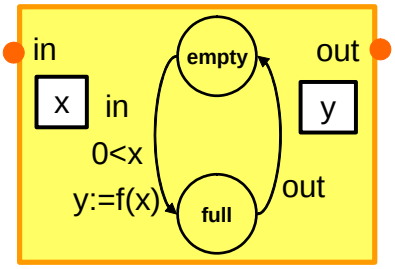


Fig. 4. An example of an atomic component in BIP.

Fig. 4 shows an example of an atomic component with two ports in, out, variables x, y, and control states $empty$, $full$. At control state $empty$, the transition labeled in is possible if $0 < x$. When an interaction through in takes place, the variable x is eventually modified and a new value for y is computed. From control state $full$, the transition labeled out can occur.

Connectors specify the interactions between the atomic components. A connector consists of a set of ports of the atomic components which may interact. If all the ports of a connector are incomplete then synchronization is by *rendezvous*. That is, only one interaction is possible, the interaction including all the ports of the connector. If a connector has one complete port then synchronization is by *broadcast*. That is, the complete port may synchronize with the other ports of the connector. The possible interactions are the non empty sublists containing this complete port. the feasible interactions of a connector and in particular to model the two basic modes of synchronization, *rendezvous* and *broadcast*.

Priorities in BIP are a set of rules used to filter interactions amongst the feasible ones.

The model of a system is represented as a BIP compound component which defines new components from existing

[3]The BIP tool-set can be downloaded from: `http://www-verimag.imag.fr/~async/BIP/bip.html`.

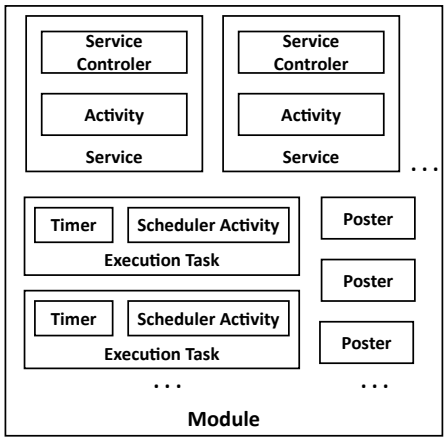


Fig. 6. A componentized GenoM module.

components (atoms or compounds) by creating their instances, specifying the connectors between them and the priorities.

The BIP framework consists of a language and a toolset including a front-end for editing and parsing BIP programs and a dedicated platform for the model validation. The platform consists of an engine and software infrastructure for executing simulation traces of models. It also allows state space exploration and provides access to model-checking tools like *Evaluator* [10]. This permits to validate BIP models and ensure that they meet properties such as deadlock-freedom, state invariants and schedulability.

The back-end, which is the BIP engine, has been entirely implemented in C++ on Linux to allow a smooth integration of components with behavior expressed using plain C/C++ code.

IV. The Functional Layer in BIP

The LAAS architecture makes use of a generic module for its functional layer. If we model this generic module and its components in BIP and if we then instantiate it and connect the existing "codels" to the resulting component, then we have a BIP model of the GenoM modules. Adding the BIP model of the interaction between the modules will give us a BIP model of the overall functional layer.

In order to formalize the componentization approach, we propose the following mapping (+ for one component or more, and . for composing components):

functional level ::= (module)+
module ::= (service)+ . (execution task) . (poster)+
service ::= (service controler) . (activity)
execution task ::= (timer) . (scheduler activity)

As shown in Fig. 5, a component modeling a generic *Service* is obtained from composing the atomic components *service controller* and *activity*. The left sub-component represents the execution task of a service. It is launched by synchronization through port *trigger*. The service controller then *controls* the validity of the parameters of the request (if available) and will either *reject* the request or *start* the activity by synchronizing with the activity component (right sub-component). In each state, the status of the execution task is available by synchronizing through port *status*. The activity will then wait for execution (i.e. synchronization on the *exec* port with the control task) and will either safely *end*, *fail*, or *abort*. Each of the transitions *control*, *start*, *exec*, *fail*, *finish* and *inter* may call an external function.

The *service* components are further composed with *execution task* and *poster* components to obtain a *module* component (See Fig. 6).

A. A Functional Module in BIP

The full BIP description of the functional level of the robot, which consists of several modules, is beyond the scope of this paper. We rather focus on the modeling of the NDD module.

The NDD module contains six *services*, a *poster* and a *control task* as sub-components and the connectors between them, as shown in Fig. 7.

The *control task* wakes up periodically (managed by the bottom-left component with alternating sleep and trigger transitions) and always triggers the *Permanent* service at the beginning of each period. During a period, the services will have authorization to execute through interactions with the control task.

Moreover, the BIP formalism allows complex relations to be defined, such as:

- interruptions, as modeled by the connector joining Stop.exec and GoTo.abort; if service Stop is executed, the GoTo algorithm will be aborted;
- constraints, as modeled by the goTo connector (in blue); service GoTo can be launched only if SetParams, SetSpeed and SetDataSource have been already completed (information available through their status port).

The BIP tool-chain generates code from the BIP model, which can be executed by the BIP engine. The code contains calls to functions from libraries originally designed for GenoM modules, which executes the real activities of the robotic system. The code generated for the NDD module has been integrated and executed. In particular, it was fully integrated with the decisional layer by replacing the functional layer originally modeled with GenoM with the one modeled in BIP.

B. Functional Level Controller Synthesis

Previously, in the LAAS architecture, a centralized controller (R2C) was used to control the proper execution of the services and to enforce the safety constraints and modules interactions. On the contrary, in the BIP model, the proper execution order and the safety properties are enforced by the BIP connectors between the controllers of different services. A BIP connector has guarded actions associated to each of its possible interactions. Dependency between the controllers of service in different modules are modeled by connectors associated with guards which represents either some valid execution condition or some safety rule. The composite behavior of these local controllers, synchronized by the

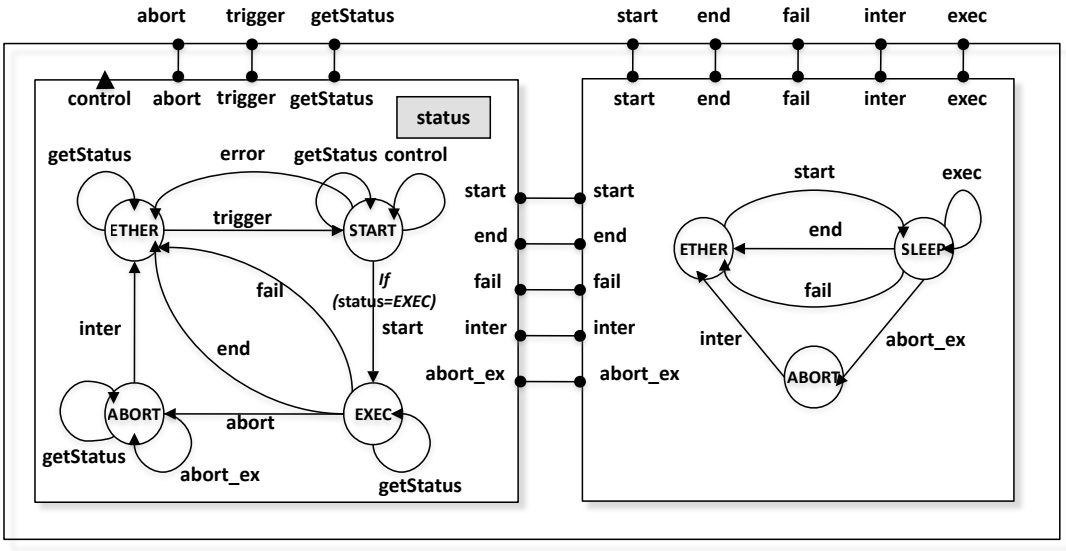


Fig. 5. BIP model of a service.

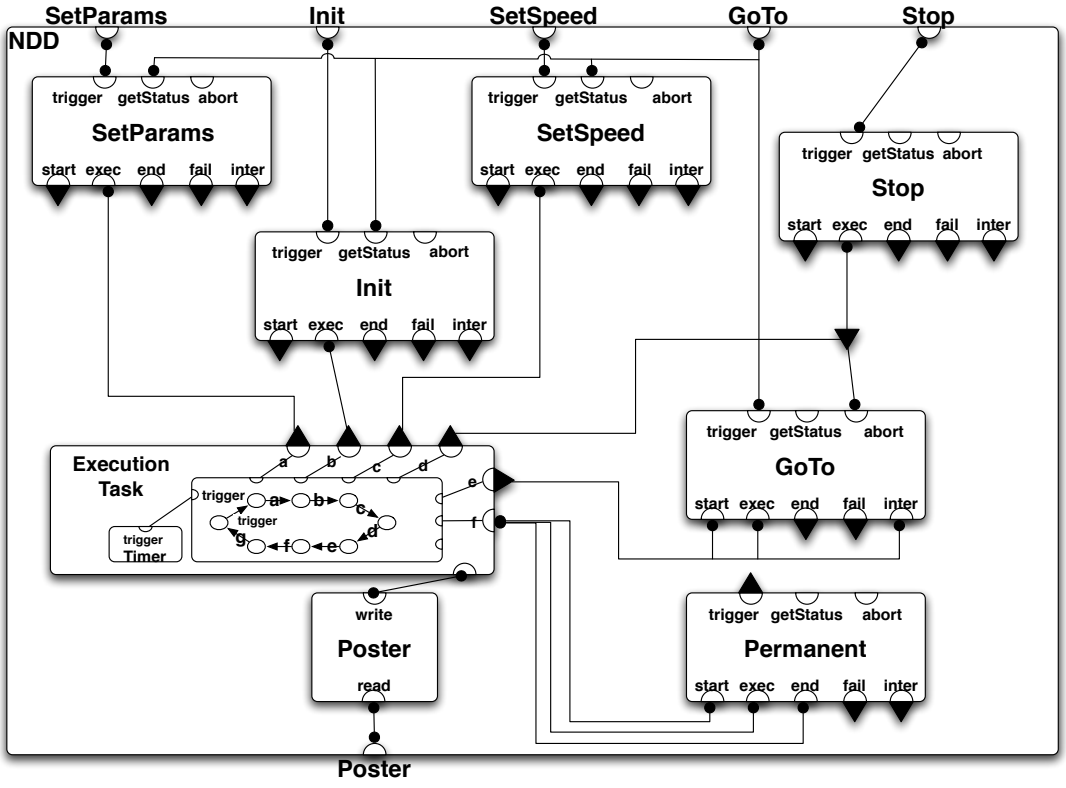


Fig. 7. The NDD module.

connectors and restricted by priorities, is equivalent to the behavior of the centralized controller.

As an example, we had to enforce a rule between the NDD and the POM modules which states that the robot can navigate using the GoTo service of the NDD module only if the module POM has already executed successfully its Run service (which updates poster `Pos`). Such a rule is enforced by constructing a connector between port *trigger* of the Goto service and port *status* of the Run service, and guarded by the *status* value.

C. Verification of Safety Properties

The BIP tool-set can perform an exhaustive state-space exploration of the system. Additionally, it can detect potential deadlocks in the system. These features have been used to verify some properties in the model of the robot and for

detection of deadlocks. Two kinds of properties have been verified.

1) Safety Properties: A safety property guarantees that something unexpected will never happen. For the verification of such properties, we used methods based on state-space exploration. The basic idea is to generate all reachable states and state changes of the system under consideration, and represent this as a directed graph called the *state-space*. Two different methods have been applied.

Model checking [15], [3] We used the model-checker tool *Evaluator* [10] which performs on-the-fly verification of temporal properties on the state-space generated by the BIP engine on exploration of the system. As an example, we describe the usage of this method in verifying a safety property of the NDD module. It is required that the *GoTo* service is triggered only after a successful termination of *SetSpeed* service. To ensure this, in the BIP model of NDD, we need to guarantee that the interaction *GoTo:trigger* occurs only after the occurrence of the interaction *SetSpeed:finish*. We checked for violations of this property, i.e finding a transition sequence in the state-space where *GoTo:trigger* is not preceded by *SetSpeed:finish*. The result obtained by *Evaluator* proves that the initialization property is preserved in the NDD module.

Verification using Observers [17], [13] For a given system S and a safety property P, we construct first an observer for P, i.e. an automaton which monitors the behavior of S and reports an error on violation of P. The verification consists of exploring the state-space of the product system. Such a method has been used to verify a timing property in the NDD module. It is needed to verify that the total time taken by all the services called within a period does not exceeds the period.

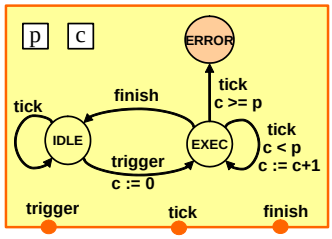


Fig. 8. Observer for the control task period verification.

In BIP, it is possible to model time as symbolic time [2] by using *tick* ports and clock variables in every timed component. Time progress is by strong synchronization of all the *tick* ports. The clock variables are incremented on a *tick*, to model function execution times. Fig. 8 shows the observer component used to verify the timing property of the NDD module. It has a clock variable c and a parameter p representing the period of the *control task*. It synchronizes with the *control task* and tracks the cumulative time taken by the services triggered by *control task*. If this time exceeds the period p, the observer moves to the *ERROR* state. During exploration, if a global system state, containing the *ERROR* state of the observer is reachable, then the property is violated.

2) Deadlock Freedom: This is an essential correctness property as it characterizes a system's ability to perform some activity over its life time. The BIP toolset allow detection of potential deadlocks by static analysis of the connectors in the BIP model [7]. It generates a dependency graph and for each cycle in this graph, a boolean formula is generated. The satisfiability of the formula is then checked by the tool *minisat* [4], where a solution corresponds to a potentially deadlocked global state. Presence of an actual deadlock can then be verified by reachability analysis of the deadlocked states, starting from the initial state of the system. The analysis for the NDD module found a potential deadlock for the state where all services are in the *EXEC* state, all activities are in the *ETHER* state, and the control task is in the Q_0 state. However, this state is unreachable, hence the deadlock is not possible.

V. State of the Art, Current Results and Prospective

The design and development of autonomous robots and systems is a very active research field. There are other architectures addressing similar problems: to provide an efficient, reusable and formally sound organization of robot software. CLARAty [12], used on various NASA research rovers, provides a nice object oriented hierarchical organization over two layers, but there is no formal model of the component interactions, nor modules canvas. IDEA [5] and T-REX [11], developed at NASA Ames and MBARI, have an interesting modular/component organization with a temporal constraint based formalism. However, complexity of constraint propagation is an obstacle for effective deployment on real-time functional modules. RMPL [9], [18] and its associated tools, propose a system based on a model-based approach. The programmers specify state evolution with invariants expressed in an "Esterel like" language and a controller maintaining them.

In [8], the authors present the CIRCA SSP planner for hard real-time controllers. This planner synthesizes off-line controllers from a domain description and then deduce the corresponding timed automata to control the system on-line. These automata can be formally validated with model checking techniques. However, this work focuses on the decisional part of the overall architecture. In [16] the authors present a system which allows the translation from MPL (Model-based Processing Language) and TDL (Task Description Language) to SMV, a symbolic model checker language. Compared to our approach, this does not address componentization and is designed for the high level specification of the decisional level.

The paper presents an approach integrating component-based construction and validation of robotic systems. It shows that a complex robotic system can be considered as the composition of a small set of atomic components. Even if we build up on the pre-existing modular LAAS architecture for autonomous robots, and model in BIP all the generic components of this architecture, such an approach could be

used with other robot software architectures and tools. The approach has been implemented and we now have a BIP controller for a subset of the functional layer of DALA, running in simulation and on the robot. The paper shows that it is possible to combine standard verification techniques, based on global state exploration, with structural analysis techniques for deadlock detection. A useful work direction is the online monitoring of the functional level execution using observer components, which would be able to generate feedback actions for the decisional level which can be useful for error-recovery. Another work direction is to extend the BIP model to take into account the decisional capabilities of autonomous systems (action planning, execution control).

REFERENCES

[1] R. Alami, R. Chatila, S. Fleury, M. Ghallab, and F. Ingrand. An architecture for autonomy. *IJRR, Special Issue on Integrated Architectures for Robot Control and Programming*, 17(4), 1998.

[2] A. Basu, M. Bozga, and J. Sifakis. Modeling heterogeneous real-time components in BIP. In *SEFM*, Pune, India, 2006.

[3] E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Workshop on Logic of Programs*, Yorktown Heights, NY, USA, 1981.

[4] N. Eén and N. Sörensen. An extensible SAT−solver. In *SAT*, Portofino, Italy, 2003.

[5] A. Finzi, F. Ingrand, and N. Muscettola. Robot action planning and execution control. In *IWPSS*, Darmstadt, Germany, 2004.

[6] S. Fleury, M. Herrb, and R. Chatila. GenoM: A tool for the specification and the implementation of operating modules in a distributed robot architecture. In *IROS*, Grenoble, France, 1997.

[7] G. Goessler and J. Sifakis. Component-based construction of deadlock-free systems. In *FSTTCS*, Bombay, India, 2003.

[8] R. P. Goldman and D. J. Musliner. Using model checking to plan hard real-time controllers. In *AIPS Workshop on Model-Theoretic Approaches to Planning*, Breckenridge, CO, USA, 2000.

[9] P. Kim, B. C. Williams, and M. Abramson. Executing reactive, model-based programs throgh graph-based temporal planning. In *IJCAI*, Seattle, WA, USA, 2001.

[10] R. Mateescu and M. Sighireanu. Efficient on-the-fly model-checking for regular alternation-free mu-calculus. Technical Report 3899, INRIA Rhône-Alpes, France, 2000.

[11] C. McGann, F. Py, K. Rajan, H. Thomas, R. Henthorn, and R. McEwen. T-REX: A deliberative system for AUV control. In *ICAPS WS on Planning and Plan Execution for Real-World Systems*, Providence, RI, USA, 2007.

[12] I.A. Nesnas, A. Wright, M. Bajracharya, R. Simmons, and T. Estlin. CLARAty and challenges of developing interoperable robotic software. In *IROS*, Las Vegas, NV, USA, 2003.

[13] M. Phalippou. Executable testers. In *IWPTS*, Tokyo, Japan, 1994.

[14] F. Py and F. Ingrand. Dependable execution control for autonomous robots. In *IROS*, Sendai, Japan, 2004.

[15] J-P. Queille and J. Sifakis. Specification and verification of concurrent systems. In *Int. Symposium on Programming*, Torino, Italy, 1982.

[16] R. Simmons, C. Pecheur, and G. Srinivasan. Towards automatic verification of autonomous systems. In *IROS*, Takamatsu, Japan, 2000.

[17] J. Tretmans. A formal approach to conformance testing. In *IWPTS*, Tokyo, Japan, 1994.

[18] B. C. Williams, M. D. Ingham, S. Chung, P. Elliott, M. Hofbaur, and G. T. Sullivan. Model-based programming of fault-aware systems. *Artificial Intelligence*, 24(4), 2003.